



Inference in infectious disease systems practicals

Contents

Acknowledgements	7
Structure of this workshop	9
Tasks	9
Lecture Notes	9
I Introduction to R	11
1 Getting Started in R	13
1.1 Installing R and RStudio	13
1.2 What is R?	14
1.3 RStudio	14
1.4 Setting up an R session	14
1.5 Console Pane	15
1.6 Script pane and R scripts	17
1.7 R packages	18
2 Fundamentals of R	23
2.1 Variables in R	23
2.2 Functions in R	24
2.3 Help files	26
2.4 Objects in R	28
2.5 Reading in data	44
2.6 Saving outputs	47
II Programming	49
3 Programming—Practical 1	51
3.1 Getting started	51

3.2 Predicting the behaviour of for loops	51
3.3 Writing a <code>for</code> loop	53
3.4 Writing a <code>while</code> loop	53
3.5 Using loops to do more complex calculations	53
3.6 Writing simple functions	54
3.7 Predicting the behaviour of a function	55
3.8 Return values from functions	55
3.9 Additional tasks for those with programming experience	56
4 Programming—Practical 2	63
4.1 Random variables	63
4.2 Probability distributions	63
4.3 Using probability distributions in R—types	65
4.4 Example—using probabilities for simulation	71
4.5 Example—multiple choices	73
4.6 Example—using R to model a death process stochastically	74
III Dynamic models	77
5 Dynamic models—Practical	79
5.1 A bit of theory	79
5.2 The return of the logistic model	81
5.3 The revenge of the amoeba	84
IV Epidemic models	87
6 Epidemic models—Practical 1	89
6.1 The basic <i>SIR</i> model	89
6.2 Coding the model	89
6.3 Some properties of the model	92
6.4 Calculating the infectious period	95
6.5 Optional section—extending <i>SIR</i> to other compartments	95
6.6 Appendix—Epidemic peak and final epidemic size	96
7 Epidemic models—Practical 2	99
7.1 Outline	99
7.2 Vaccination at birth	99
7.3 Targeted Vaccination	101

8 Epidemic models—Practical 3	105
8.1 Stochastic models	105
8.2 Coding a simple <i>SI</i> model	105
8.3 Dissecting the stochastic <i>SI</i> code	107
8.4 Performing multiple simulations	109
8.5 Exploring the effects of stochasticity	112
8.6 The delay time	113
8.7 Coding a simple <i>SIR</i> model	116
8.8 <i>SIS</i> Extension: Adding Recovery from Infection	120
8.9 <i>SIS</i> Extension: Adding host demography	120
8.10 <i>SIR</i> Extension: The final epidemic size	121
8.11 <i>SIR</i> Extension: Stochastic extinction	123
V Statistical modelling	125
9 Estimating R_0 using catalytic models	127
9.1 Estimating the force-of-infection and R_0 of rubella	127
9.2 Next-generation matrix approaches	135
9.3 Estimating the next generation matrix from rubella serology	136
10 Particle MCMC	143
10.1 Case Study	143
10.2 Arguments for PMCMC() function	145
10.3 Running the PMCMC algorithm	148
10.4 Optimising the number of particles	151
10.5 Visualising and summarising the posterior distributions	155
10.6 Predictive posterior distributions	157
VI Additional Materials	161
11 Estimating R_0	165
11.1 Introduction: models and data sets	165
11.2 Final size method	170
11.3 Regression method	172
11.4 RECON earlyR method	178
11.5 Appendix	180

12 Seasonality and Measles Epidemics	183
12.1 Measles data and challenge	183
13 Outbreak of influenza in a boarding school	189
13.1 Data summary and challenge	189
VII Appendices	191
References	193
Answers	195
Additional information	265

Acknowledgements

The materials, practicals and examples for this course were originally developed for the short course “Mathematical Models for Infectious Disease Dynamics” organised and run by the Wellcome Trust Advanced Courses Program between 2010–2020. A number of instructors, in addition to those involved in the current iteration, have contributed and adapted these materials over the years as the course has evolved. We would like to thank and acknowledge the contribution of all of these course alumni:

- Dr Olivier Restif ([Department of Veterinary Medicine, University of Cambridge](#))
- Dr Andrew Conlan ([Department of Veterinary Medicine, University of Cambridge](#))
- Dr TJ McKinley ([Medical School, University of Exeter](#))
- Dr Cerian Webb ([Department of Plant Sciences, University of Cambridge](#))
- Dr Ken Eames (Church of England Research and Statistics unit)
- Dr Nik Cunniffe ([Department of Plant Sciences, University of Cambridge](#))
- Dr Matt Castle ([Cambridge Centre for Data-Driven Discovery, University of Cambridge](#))
- Dr Ellen Brooks Pollock ([School of Social and Community Medicine, University of Bristol, UK](#))
- Dr Leon Danon ([Department of Engineering Mathematics, University of Bristol](#))
- Dr Katy Gaythorpe ([Faculty of Medicine, School of Public Health, Imperial College](#))
- Dr Roberto Sainz (Facultad de Ciencias, Universidad de Colima)
- Dr Joshua Ross ([School of Computer and Mathematical Sciences, University of Adelaide](#))
- Dr Robin Thompson ([Mathematical Institute, University of Oxford](#))
- Dr Sophie Ip
- Dr Johann von Kirchbach
- Dr James Cox

Structure of this workshop

In these notes, R commands to be entered into the console are shown in grey boxes e.g.

```
seq(0, 1, length = 11)
```

and the corresponding outputs look like:

```
## [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Tasks

Task 1

All **tasks** will be denoted in panel boxes like this one. In the HTML version, all solutions can be toggled by hitting the **Show Solution** buttons. In the PDF version solutions are given in the Appendix and are linked via the **Show Solution** buttons.

Show: Solution on P195

Lecture Notes

Note

The lecture notes for all parts of this workshop can be downloaded [here](#).

Part I

Introduction to R

Chapter 1

Getting Started in R

TJ McKinley (t.mckinley@exeter.ac.uk)

R—available at <https://cran.r-project.org/>—is a comprehensive statistical programming language. It can be thought of as an open-source, freely available implementation of the S language. R is an **interpreted language**, and so can be used interactively, without having to compile the code into an executable file.

Although R is a functional programming language in its own right, its popularity is in large part due to its fantastic capabilities as a statistical package. However, R is much more than that. R has amazing graphical capabilities; can integrate with other languages, such as C and Python; can use scripts and be run in batch mode; and can even be used to produce **reproducible documents**, presentations and **interactive webpages**. Even better than that, R is completely **free** and **open-source**!

To top it off, R is also **multi-platform**, and so can be used on Windows, Mac and Linux operating systems. R is supported by a wide user base, and is extended by a large number of additional packages, and even provides the capabilities to create your own. Whilst R provides its own development environment, we will use a fantastic IDE¹ provided by RStudio. This is free to download, provides some neat features, and crucially, operates the same way on all operating systems!

All of the instructions in these practicals will assume that you are using RStudio.

1.1 Installing R and RStudio

If you want to download R for your own computer, first go to <http://cran.r-project.org>, and follow the instructions in the box called ‘Download and Install R’. **You only need to install the base package**. During installation, selecting the default options should be sufficient.

Once you have installed R, you can install the **free desktop** version of RStudio from <https://www.rstudio.com/products/rstudio/download/>. Again, selecting the default options during installation should be sufficient.

Important

You will also have to install some additional R packages in order to complete the materials in this workshop. Details on how to install packages can be found at the end of this Chapter [here](#). I would suggest you read through all these notes before installing the packages, but this box is here to remind you to do this. A full list of which packages to install can also be found [here](#).

¹Integrated Desktop Environment

1.2 What is R?

R is an **object-orientated** programming language. This means that you create objects, and give them names. You can then **do things** to those objects: you can perform calculations, statistical tests, make tables or draw plots. Objects can be single numbers, characters, vectors of numbers, matrices, multi-dimensional arrays, lists containing different objects and so on.

1.3 RStudio

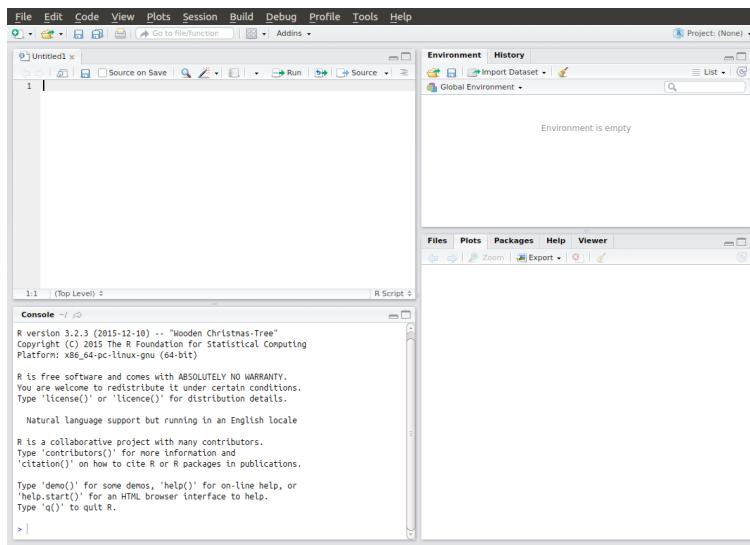


Figure 1.1: RStudio window

After loading, the RStudio window should look something like Figure 1.1. It consists of:

- **Script pane** (top-left): this is essentially RStudio's built-in text editor. It has all the usual features one would expect: syntax highlighting, automatic indentation, bracket matching, line highlighting and numbering and so on. You can open any type of text file in here, not just R scripts. (You might have to go to *File > New File > R Script* to open a new R script if you don't have one already open.)
- **Console pane** (bottom-left): This is where you run R commands and view outputs.
- **Workspace/history pane** (top-right): this shows a list of all of the objects and variables that you create during a session or a history of all of the commands that have been sent to the command window during the session.
- **Plot/help pane** (bottom-right): this shows any plots that you create or any help files that you access.

You can alter the size of the various panes by clicking and dragging the grey bar in between each window to suit your needs. You can also change their arrangement by going to *Tools > Global Options*, and then selecting the *Pane Layout* option.

1.4 Setting up an R session

It is worthwhile getting into a workflow when using R. General guidelines I would suggest are:

- Use a different folder for each new project / assignment. This helps to keep all data / script / output files in one self-contained place.
- Set the **Working Directory** for R at the outset of each session to be the folder you've specified for the particular assignment you're working on. This can be done in RStudio by going to *Session > Set Working Directory > Choose Directory*. This sets the default search path to this folder.
- Always use **script files** to keep a record of your work, so that it can be reproduced at a later date.

We will explore the **console** and **script** panes below, dealing with the other panes as and when they arise.

1.5 Console Pane

The console pane provides a direct interface with R, and looks similar to command line R (in Linux and Macs), and the console pane in R for Windows. You enter commands via the standard prompt `>`. For example, type the following into the console pane:

```
10 + 5 * 3
```

```
## [1] 25
```

You can see here that R has returned a value of 25, illustrating one of R's key features: that it can be used as an overgrown calculator, simply by entering commands into the prompt. R supports lots of basic mathematical operators, such as those found in Table 1.1.

Table 1.1: Basic mathematical operators

Symbol	Meaning
<code>+</code>	addition
<code>-</code>	subtraction
<code>*</code>	multiplication
<code>/</code>	division
<code>^</code>	to the power
<code>%%</code>	the remainder of an integer division (modulo)
<code>/%%</code>	integer division

Meanwhile, Table 1.2 has some other ones you might need. (**Note:** these are functions: just replace e.g. `x` with a number.)

Table 1.2: Other useful mathematical functions

Function	Meaning
<code>log(x)</code>	$\log_e(x)$ (or $\ln(x)$)
<code>exp(x)</code>	e^x
<code>log(x, n)</code>	$\log_n(x)$
<code>log10(x)</code>	$\log_{10}(x)$
<code>sqrt(x)</code>	\sqrt{x}
<code>factorial(x)</code>	$x!$
<code>choose(n, x)</code>	binomial coefficients: $\frac{n!}{x!(n-x)!}$
<code>gamma(x)</code>	$\Gamma(x)$ for continuous x or $(x-1)!$ for integer x
<code>lgamma(x)</code>	natural log of $\Gamma(x)$

Function	Meaning
<code>floor(x)</code>	greatest integer $< x$
<code>ceiling(x)</code>	smallest integer $> x$
<code>trunc(x)</code>	closest integer to x between x and 0 e.g. <code>trunc(1.5) = 1, trunc(-1.5) = -1</code>
<code>round(x, digits = 0)</code>	<code>trunc</code> is like <code>floor</code> for positive values and like <code>ceiling</code> for negative values
<code>signif(x, digits = 6)</code>	round the value of x to an integer
<code>cos(x)</code>	round x to 6 significant figures
<code>sin(x)</code>	cosine of x in radians
<code>tan(x)</code>	sine of x in radians
<code>acos(x), asin(x), atan(x)</code>	tangent of x in radians
<code>acosh(x), asinh(x), atanh(x)</code>	inverse trigonometric transformations of real or complex numbers
<code>abs(x)</code>	inverse hyperbolic trigonometric transformations on real or complex numbers
	the absolute value of x , ignoring the minus sign if there is one

R retains a **history** of all the commands you have used in a particular session. You can scroll back through these using the up (\uparrow) and down (\downarrow) arrows whilst in the console pane. (You can even save the history—though we will discuss a *much* better option in the next section.)

One important thing to note is that unlike a language like C, R does not require the semicolon (`;`) symbol to denote the end of each command. A carriage return is sufficient. A semicolon can be used to allow multiple commands to be written on the same line if required. For example,

```
10 + 5 * 3; sin(10)
```

```
## [1] 25
## [1] -0.5440211
```

is equivalent to

```
10 + 5 * 3
sin(10)
```

```
## [1] 25
## [1] -0.5440211
```

One thing to note is that if a command is incomplete, then R will change the `>` prompt for a `+` prompt. For example, typing `10 + 5 *` into the console pane will result in the `+` prompt appearing, telling you that the previous line is incomplete i.e.

```
> 10 + 5 *
+ 3
```

```
## [1] 25
```

You must either complete the line or hit the `Esc` key to cancel the command.

1.6 Script pane and R scripts

The console window is the engine room of R, and one can interact directly with it. One key advantage to R is that it *records* all of the commands that you enter into the console (known as the command *history*). It is possible to save the command history, or run back through it using the arrow keys. However, a much better approach is to use the **script pane** to write an R script that contains all the commands necessary for a particular project. In fact, one might argue that this is probably one of the most important features of R relative to a point-and-click statistical package such as SPSS.

Put simply, **R scripts are just text files that contain commands to run in R**. They are **vitally important** for the following reasons:

- They keep a systematic record of your analysis, which enables you to **reproduce** your work at a later date. Scripts can be passed to collaborators or other users to enable them to **replicate** your work.
- This record means that you do not have to rely on your memory to figure out **what** you did.
- R scripts allow you to **comment** your code, which means that you also won't forget **why** you did it.
- In more advanced settings, R scripts can also be run in **batch** mode, which means that you can run scripts remotely on a server somewhere without having to be sat in front of a computer manually entering commands.
- Although programs like SPSS allow **outputs** to be saved, R scripts contain **inputs**, which are much more useful, since it is easier to generate the outputs from the inputs than it is to reconstruct the likely inputs from the outputs.
- In fact, R scripts can be combined with a markup language called 'markdown' to generate fully reproducible documents, containing both inputs and outputs. It does this using the fantastic **knitr** and **rmarkdown** packages. (In fact these notes were written using **rmarkdown** and a package called **bookdown**.)

Some comments:

- RStudio comes with its own text editor, but if you are not using RStudio, then there are plenty of others available.
- R is case-sensitive. If something doesn't work, it's often because you have failed to capitalise, or capitalised where you shouldn't have. **NEVER, EVER, EVER** use Word to edit your R scripts! Word often tries to correct your grammar and is an absolute nightmare for writing code. If you don't like RStudio's editor, then lots of lightweight and free text editors exist that you can use. By all means use Word for writing up the work², but **please, NEVER** use it for writing code!

In RStudio, you can open a new script in R using: *File > New File > R Script*.

Type the following into the **script file**:

```
## calculate the hypotenuse from a right-angled triangle
## with the two other sides equal to 3 and 4
sqrt(3^2 + 4^2)
```

Notice that nothing has happened. All you've done is write some commands into a text window. However, if you highlight these lines and then hit the **Run** button in the top right-hand corner of the script pane (or, in Windows certainly, press **Ctrl-Enter**), then RStudio runs these lines in the console pane. (Alternatively, you can manually copy-and-paste these lines into the console window.) This should return:

```
## [1] 5
```

²although **rmarkdown** provides a powerful alternative

Note

Notice that the `#` symbol in the input code denotes a **comment**, such that any text after the `#` is ignored (up to the end of the current line). I have used a double hash here `##`, though this is simply because I've written this practical using [R Markdown](#) and it seems to typeset better. This is not absolutely necessary though, since anything after the first `#` is ignored.

Comments are **vital** in code to ensure **reproducibility** and **readability**. You should ensure that all code is commented, so that both you and anyone else who wants to use your code is able to decipher it. Even if no one else is going to look at your code, it is still worthwhile to comment it. What seems obvious to you as you write a piece of code, often becomes confusing when you return to it in six months time and can't remember what you did or why you did it...

It is conventional to save R script files using the suffix '`R`', though remember that they are simply **text files**, and can be viewed in any text editor. Make sure you have set up a folder to store the code for this practical in, and have changed the **working directory** to this folder (see the [setup](#) section), and then save this script file as something like `IntroToR.R`.

Important

Make sure you save your script file regularly to prevent data loss!

1.6.1 Notes on legibility

NOTE: that I use spaces within the code to make it clearer. R does not require this, but again I think it is good practice to think about how to make your code legible. Different coders have different preferences, but personally I prefer `plot(Worm.density ~ Vegetation, data = worms)` over `plot(Worm.density~Vegetation,data=worms)`. As [Hadley Wickham](#) says: "Good coding style is like correct punctuation: you can manage without it, but it sure makes things easier to read."

Note that different coders prefer different styles—there is no universal agreement, and you will notice that different lecturers on this course have slightly different styles. However, it's worth getting into the habit of writing your code neatly and with legibility in mind. A guide that is similar to my own is the **tidyverse** guide [here](#). (Note that unlike e.g. Python, R does not require specific indentation. Instead it uses curly brackets to group lines of code together. However, indentation is still key for legibility.)

I tend to use script files to keep a record of all commands that need to be reproduced, but I often enter commands directly into the console window when I'm testing or visualising things. Try to keep redundant code out of your script files and try to make sure the code is legible and commented.

1.7 R packages

R has hundreds of add-on **packages** that provide functionality for a wide range of techniques. These repositories are growing all of the time; some packages become redundant and are removed, others are updated, some are superceded or incorporated into others, and completely new ones appear regularly. A key part of becoming proficient with R is learning how to install and update packages.

Note

R packages can be thought of in a similar way to Matlab toolboxes or Python libraries.

The principal R package repository can be found on [CRAN](#) (the Comprehensive R Archive Network). Another popular repository, predominantly aimed at bioinformatics packages, is [Bioconductor](#), though installation of packages through Bioconductor is more difficult than through CRAN, so we will focus on the latter only here.

Some packages are included as part of R's base package. To load a package, you can use the `library()` or `require()` functions, passing the name of the required package. For example, to load the `tidyverse` package, type:

```
library(tidyverse)
```

If this doesn't return any error, then the package is loaded and you are now able to use any function in `tidyverse` in your R code. R packages must contain help files and documentation in order to be included on CRAN. For example, the documentation for the `tidyverse` package can be found [here](#), through the *Reference Manual* link, plus some vignettes through the *Vignettes* link.

1.7.1 Installing packages from CRAN

If the package you want is not installed, then you will need to install it. To install a package that is hosted on CRAN, you can use the `install.packages()` function. For example, to install `tidyverse` you can type:

```
install.packages("tidyverse")
```

This will ask that you select a mirror repository—choosing one close-to-home is a good idea. It might also ask you to set up a local R library in your user directory. This is a good idea, so I would just accept the default if it asks.

If it installs without any errors, then you can load the library using `library(tidyverse)` as above.

Note

You only have to install a package **once** (unless you update R). You have to **load** the library once during each **session**. I prefer to enter all my calls to `library()` at the top of my script file, so I can quickly see which packages are required for my script to run.

1.7.2 Installing from a local file

You can also install a package from a local ZIP file if required (though this is required much less often). For example, from the [CRAN](#) website, click on the *Packages* link, find the correct package and simply download a suitable Windows / Mac binary file as appropriate, or the 'Package Source' file (if you are on Linux, or if you have the appropriate compiler tools installed). See Figure 1.2 for an example of where these can be found for the `tidyverse` package.

Once you have downloaded one of these, you can install using:

```
install.packages("PATH/TO/PACKAGENAME", repos = NULL)
```

where `PATH/TO/PACKAGENAME` is the path to the package source file (be careful to get the path in the correct format—Windows users might have to use \ instead of /). The `repos = NULL` argument tells R to look for the file locally, and not online.

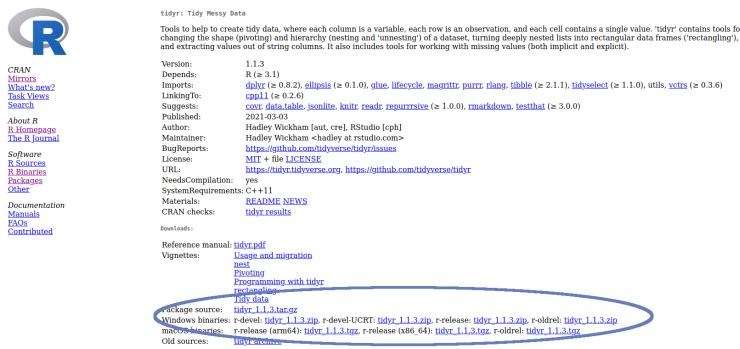


Figure 1.2: Location of source / binary files for ‘tidyr’

1.7.3 Installing archived versions of a package

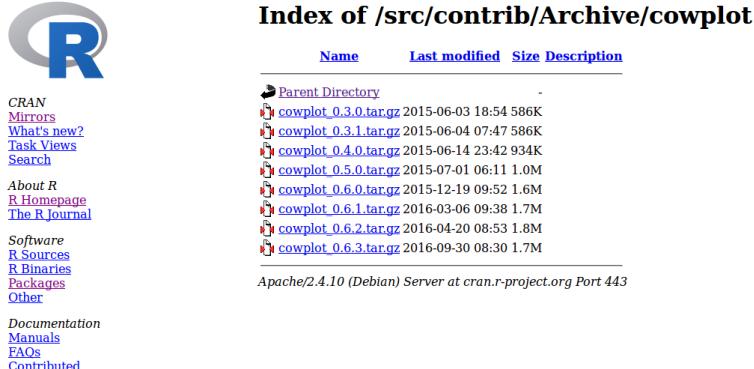
Sometimes R packages are not updated as quickly as R itself, and occasionally you might need to install an older version of a package. A really useful package is called `devtools`. This provides functions to install R packages directly from CRAN archives, which can be found by going to <https://cran.r-project.org/>, and clicking on the *Packages* link followed by the *Archived* link. This provides a list of older versions of a package, which can be installed using the `install_version()` function in `devtools` (as detailed below).

For example, I once had an issue with a package called `cowplot`, that didn't exist for my version of R, and returned the following error message (**don't run the code now**):

```
install.packages("cowplot")
```

```
Installing package into ‘/home/tj/R/x86_64-pc-linux-gnu-library/3.2’
(as ‘lib’ is unspecified)
Warning in install.packages :
  package ‘cowplot’ is not available (for R version 3.2.3)
```

I solved this by finding an older version of the package on CRAN (by clicking on *Packages* then *Archived*):



I noted the version number, and then used `install_version()` from the `devtools` package as follows (**don't run the code now**):

```
library(devtools)
install_version("cowplot", version = "0.6.3", repos = "https://cran.r-project.org/")
```

These can also be installed manually from source as in the previous section.

1.7.4 Installing development packages

In order to upload a package to CRAN, the package must pass a series of tests. The versions you find on CRAN are the **stable** versions (i.e. they have passed these tests and work well with the current version of R). However, developers often keep the latest **development** version of their package on an online repository such as [GitHub](#). Development packages usually contain the most up-to-date functions, but are likely to contain more bugs—use them at your own risk! In addition, some packages simply aren't available on CRAN—there is no requirement to add them to this repository.

For example, to install the development version of the `gganimate` package from GitHub, locate the source code on e.g. GitHub; in this case it can be found at <https://github.com/dgrtwo/gganimate>. Then the package can be installed using `devtools` as follows (**don't run the code now**):

```
install_github("dgrtwo/gganimate")
```

Note the syntax `DEVELOPER/PACKAGENAME`—this can be found from the address of the repository e.g. <https://github.com/dgrtwo/gganimate>. For CRAN packages, the source code URL is usually listed on the package page. For example, Figure 1.2 shows that the GitHub page for `tidyverse` is at <https://github.com/tidyverse/tidyverse>.

1.7.5 Packages to install for this module

The packages you need to install for this module are listed below. Some are installed automatically by R, but have been listed here for completeness. (Please let me know if you come across any that aren't listed here.)

- `deSolve`
- `earlyR`
- `MCMCpack`
- `outbreaks`
- `SimInf`
- `SimBIIID`
- `utils`
- `tidyverse`
- `incidence`
- `patchwork`
- `tsiR`
- `chron`

Many of these packages depend on other packages, but R should install all dependencies automatically. These should all be available via CRAN, and so can be installed using e.g.

```
install.packages("deSolve")
```

If you're on a Windows computer then you'll need to do some additional work to use `SimInf` and `SimBIIID` (macOS and Linux users should be fine).

- Download Rtools from <https://cran.r-project.org/bin/windows/Rtools/>
 - Click on the appropriate version of RTools on the left.
 - For RTools 4.4 (the most up to date version) click on the link `Rtools44_installer`. It's in the third paragraph down in the second section.
 - Once it's downloaded install it using all of the default settings.
 - This might take a couple of minutes, but click Finish when it's done.

Chapter 2

Fundamentals of R

TJ McKinley (t.mckinley@exeter.ac.uk)

2.1 Variables in R

R makes use of symbolic variables, i.e. words or letters that can be used to represent or store other values or objects. We will use the assignment operator `<-` to ‘assign’ a value (or object) to a given word (or letter). Run the following commands to see how this works (don’t worry about the comments, these are for your understanding):

```
## assign to x the value 5
x <- 5

## print the value assigned to the
## variable x to the screen
x

## [1] 5

## we can also assign text to a variable
y <- "Hello There"

## print y
y

## [1] "Hello There"

## we can re-assign variables
y <- sqrt(10)

## or assign a variable in terms
## of other variables
ziggy <- x + y

## print variable "ziggy"
ziggy
```

```
## [1] 8.162278
```

You will notice that as we create each of these variables, they begin to appear in the **environment** pane in the top right-hand side of the RStudio window. This shows the current R **workspace**, which is a collection of objects that R stores in memory. We can remove objects from the workspace using the `rm()` function e.g.

```
## remove the variables x and y
rm(x, y)
```

Notice that `x` and `y` have now disappeared from the workspace. The variable `ziggy` still contains the correct answer though (these are not **relative** objects, such as the macros assigned in a program like Excel).

```
ziggy
```

```
## [1] 8.162278
```

Another way of visualising the working directory is to use the `ls()` function, which lists all objects currently in the workspace e.g.

```
ls()
```

```
## [1] "ziggy"
```

Note

Names of variables can be chosen quite freely in R. They can be built from letters, digits and other characters, such as `_`. They can't, however, start with a digit or a `.` followed by a digit. Names are **case sensitive** and so `height` is a different variable from `Height`. Try to choose **informative** variable names where possible (`height` is more informative than `x` when talking about heights for example).

Don't use spaces in variable names. Underscores are much safer^a. Try to avoid using long words (since you will have to type them out each time you want to access the corresponding elements—although RStudio does have *autocomplete* functionality). Often I will try to convert as much to lower case as possible. If you do need to include multiple words try to use an underscore `_` (e.g. `child_heights`), or some people prefer to use something like **CamelCase** (e.g. `childHeights` or `ChildHeights`). Use whatever you prefer but try to be **consistent**.

^asome people also use dots, e.g. `child.heights`, but in general I would avoid this, since dots are often used when working with functions of specific classes such as `print.lm()`

2.2 Functions in R

Next we need to understand how to do more complicated things in R. For this we need functions. This section illustrates how R's inbuilt functions work. Most commands that are used in R require the use of functions. These are very similar to mathematical functions (such as `log(x)`), in that they require both the name of the function (i.e. `log`) and some **arguments** (i.e. `x`). However a lot of functions in R have multiple arguments which can be of completely different types.

The format is that a function name is followed by a set of parentheses containing the arguments. For example, if we type `seq(0, 3, 0.5)`, the function name is `seq` and the specified arguments are 0, 3 and 0.5. `seq()` is an inbuilt R function that generates sequences of numbers, and we're going to continue to use it to illustrate how functions work. Try typing in the command:

```
seq(0, 3, 0.5)
```

```
## [1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0
```

You can see that this function produces a sequence that starts at 0, ends at 3 and has steps of 0.5. Spaces before the parentheses or between the arguments are not important when writing functions so `seq(0,3,0.5)` will work just as well as `seq(0, 3, 0.5)`. However, as stated [earlier](#), I think spaces in sensible places can make the code much more legible i.e. `seq(0, 3, 0.5)` is better than either of the previous options.

All functions in R have **defaults** for the majority of their arguments (some functions have dozens of arguments, and having default values avoids you having to specify them all every time you use the function). For example, the `seq()` function has the following four main arguments (in this order):

Argument	Description
<code>from, to</code>	the starting and (maximal) end values of the sequence. Of length 1 unless just <code>from</code> is supplied as an unnamed argument.
<code>by</code>	number: increment of the sequence.
<code>length.out</code>	desired length of the sequence. A non-negative number, which for seq and seq.int will be rounded up if fractional.
<code>along.with</code>	take the length from the length of this argument.

When we write `seq(0, 3, 0.5)` R assumes that the first argument corresponds to `from`, the second argument corresponds to `to` and the third argument corresponds to `by`. This is known as **positional matching**. This means that `seq(0, 3, 0.5)` will produce a vector consisting of the sequence of numbers starting at 0, ending at 3 by adding 0.5 each time. If we write `seq(10, 0, 0.5)`, then R will attempt to create a sequence starting at 10, ending at 0 by adding 0.5 each time.

Task 2

Try typing `seq(10, 0, 0.5)`. R returns an error. Why?

Show: Solution on P195

The other type of matching is called **named matching** and this would allow you to write `seq(from = 0, to = 3, by = 0.5)` or `seq(to = 3, from = 0, by = 0.5)` as equivalent function commands (i.e. the **order** of the arguments **doesn't matter** if using *named* matching).

Using named matching also allows us to access the other argument: `length.out` if we wanted to use it, though you should recognise that you cannot, in general, use all four arguments in this case to specify a vector. (Why not, do you think?)

In fact, this command needs three arguments to work, and it doesn't matter which three of the four arguments you use. To find out what arguments a function can take and what they mean use the `?FUNCTION` syntax (where you replace **FUNCTION** with whichever function you're interested in e.g. `?seq` or `?atan`). This will open the internal R help file for the function. The [Help](#) section of these notes contains detailed information on these help files.

Being able to access these internal help files will be of particular help for all of the statistical functions we will want to use which generally have many arguments.

Task 3

Use the R function `seq()` to create the following sequences:

1. 2, 4, 6, ..., 30.
2. A sequence of 14 numbers starting at -2.5 and ending at 15.34.
3. A sequence of 7 numbers starting at 0, increasing in increments of 0.04.
4. A sequence starting at 101, ending at -20, in decrements of 11.

Show: Solution on P195

2.2.1 Nesting functions

A useful feature of R is its ability to **nest** functions. This can help make the code more concise and avoids the need to create lots of temporary variables. Nonetheless, if used too extensively then this can render code difficult to interpret, so a balance is necessary. For example,

```
exp(sqrt(10))
```

```
## [1] 23.62434
```

This is a simple nested function, since `sqrt(10)` produces a single number, which is then used as an argument to the `exp()` function. Mathematically, this gives $e^{\sqrt{10}}$. We could have also done this in two steps, but this would have involved creating a temporary variable to store the result of the sum:

```
x <- sqrt(10)
exp(x)
```

```
## [1] 23.62434
```

To understand nested functions, think about evaluating the **inner** functions first, seeing what they output and then how that slots into the next function. We will see more examples of this later.

2.3 Help files

It is a requirement of R packages that any function made available to an end user as part of a package must be documented in a **help file**. As such, the `? command` is simply a way to get more information on any given function. For example, if you wanted to know more about how to use the `hist` function (this function is used for plotting histograms), then you could type:

```
?hist
```

This will produce something like Figure 2.1 to appear in the bottom-right pane.

This new window contains all of the information you need to understand how to use the `hist()` function. Although these help files can seem daunting at first, once you get used to them they will provide the route for your continued exploration of R. Help files provide all of the details that the author of the command

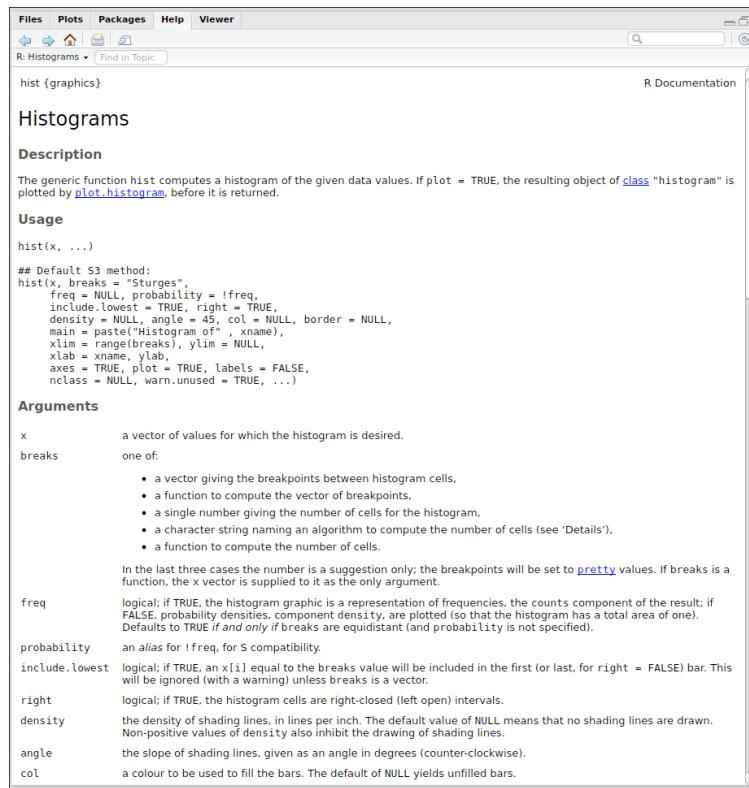


Figure 2.1: Help file for the 'hist()' function

saw fit to include. They all aim to be comprehensive but some authors assume that you are more familiar with what's going on than others. Most help files will have the following sections:

Table 2.2: General structure of R helpfiles

Section	Description
Usage:	This will show you the minimum required arguments and also what the default value are for each of the optional arguments. In this case only <code>x</code> is a required argument and all of the others are optional.
Arguments:	This gives an explanation of what each argument does and what sort of data type it should be. In this case the argument <code>x</code> needs to be a numerical vector.
Details:	This section tells you what the function should do along with any subtleties in its usage or clarifications.
Value:	This tells you what actually happens when you run the function and specifically what sort of R object the function produces. In this case it produces a special <code>histogram</code> object (which is actually just a normal list with certain specific components).
References:	This gives you some selected text books that the author feels explains the theory behind the function in greater detail.
See Also:	This should be a list of other R commands that are related to the current function. They might be of more use to you, or their help files might be better written and clarify what you're trying to do.
Examples:	This should be a set of R commands that you can copy-and-paste into either a script file or the console pane. These should give you a practical example of how to use the function in several different settings.

If you're really stuck, there is an international community of R users who regularly post questions and answers on advanced topics on sites like [StackOverFlow](#). I tend to succeed mostly by Googling the problem. For example, Googling 'R importing data' finds me this rather excellent tutorial: <http://www.r-tutor.com/r-introduction/data-frame/data-import>.

2.4 Objects in R

R defines various types of object (and allows you to define your own if you so wish). We will explore the most common object types below.

2.4.1 Vectors

Vectors are fundamental R objects. In fact, we have already seen them at work in earlier examples. For instance, notice that if we type the number 10 into the command prompt, the output when R prints this to the screen is prefixed with a [1]:

```
10
```

```
## [1] 10
```

This shows that the output is a **vector**. (The [1] in the output shows that the object is of one **dimension**. Confusingly, the 1 corresponds to the element of the vector that is first printed to the screen—so here 10 is element 1 of the vector. The fact that it has one dimension can be seen by the fact that there is only one number in the square brackets. A 2-dimensional object like a matrix, would have e.g. [1,], as we will see later.) If we had a long vector, e.g.

```
seq(1, 100, by = 2)
```

```
## [1] 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
## [26] 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99
```

then when we print to the screen the vector is split up, and we can see that the first element is 1, and the twenty-sixth element is 51 etc.

The `ziggy` variable we created earlier is also a vector of length 1:

```
ziggy
```

```
## [1] 8.162278
```

Note that vectors do not have to be numerical in value e.g.

```
"somewords"
```

```
## [1] "somewords"
```

is also a vector of length 1. R distinguishes between different 'types' of vector, with the four main types being:

- **numeric** where each element is a number (either integer or decimal);
- **character** where each element is a word or letter;
- **factor** where each element is a word or letter, but with some additional “grouping” information;
- **logical** where each element is either **TRUE** or **FALSE**.

In this section we will look at several different commands which produce vectors and then see how we can manipulate these objects in R.

2.4.1.1 Creating vectors

Probably the most common ways of creating vectors are to use the **c()**, **seq()** and **:** commands. Of these, **:** creates integer sequences, **seq()** creates structured sequences (only useful for numeric vectors), and **c()** is short-hand for **concatenate**, and it binds together all of the elements that you put inside the brackets to make a single vector.

Run the following commands to see how we can create different types of vectors:

```
1:5
6:2
```

```
## [1] 1 2 3 4 5
## [1] 6 5 4 3 2
```

Notice that brackets can be important, for example, note the differences below.

```
-4:1
-(4:1)
```

```
## [1] -4 -3 -2 -1  0  1
## [1] -4 -3 -2 -1
```

The first line above creates an integer sequence from -4 to 1, and the second creates one from -4 to -1. The **seq()** command we saw earlier, and here are some more examples:

```
seq(0, 1, by = 0.1)
seq(0, 1, length = 6)
seq(10, 5, by = -0.5)
```

```
## [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
## [1] 0.0 0.2 0.4 0.6 0.8 1.0
## [1] 10.0 9.5 9.0 8.5 8.0 7.5 7.0 6.5 6.0 5.5 5.0
```

The **c()** function is very useful for concatenating non-sequential numbers or vectors of other data types e.g.

```
c(1, 3, 5, 7)
c(0.1, -6, 1, 0)
c("bob", "sam", "ted")
```

```
## [1] 1 3 5 7
## [1] 0.1 -6.0 1.0 0.0
## [1] "bob" "sam" "ted"
```

Note

Notice that we needed to use double-quotes to signify `character` types e.g. `c("bob", "sam", "ted")`. Note that R does not distinguish between single or double-quotes, so `c('bob', 'sam', 'ted')` would have worked equally well.

If you forget the quotes (e.g. `c(bob, sam, ted)`) then R would instead have looked for three **objects** called `bob`, `sam` and `ted` and tried to concatenate these together. If these objects do not exist, then R will return an error:

```
c(bob, sam, ted)
```

```
## Error in eval(expr, envir, enclos): object 'bob' not found
```

To this end, it is worth noting that we can also use `c()` to concatenate objects that are of the same type together. For example:

```
c(1, -6, ziggy)
```

```
## [1] 1.000000 -6.000000 8.162278
```

```
c(ziggy, seq(3, 6, length = 5))
```

```
## [1] 8.162278 3.000000 3.750000 4.500000 5.250000 6.000000
```

(In the example above we have used **nested** `c()` functions to make the code more concise.) If we try to concatenate objects of a different type, R will either return an error, or it will convert one of the objects into the same class as the other. For example,

```
c(c("bob", "sam", "ted"), ziggy)
```

```
## [1] "bob"           "sam"           "ted"           "8.16227766016838"
```

(Here we use **nested** `c()` functions to first create a `character` vector out of the strings `"bob"`, `"sam"` and `"ted"`, and then to concatenate this with the object `ziggy`.) Notice that the concatenated vector is a `character` vector (notice the **double-quotes** around `"8.16227766016838"`), indicating that R has converted the elements of `ziggy` into `characters` in order to concatenate them with a `character` vector.

Note

R will always convert to the most complex object type: so a combination of `character` and `numeric` values will be converted to a `character`.

Notice that R has not changed the value of the object `ziggy`—if we print `ziggy` to the screen we can see it is still a `numeric` vector:

```
ziggy
```

```
## [1] 8.162278
```

Rather, it created a copy of `ziggy` which it converted into a `character` before concatenating. In order to change the value of an object permanently, we need to reassign the object. For example, the function `as.character()` will convert a `numeric` vector into a `character` vector without changing the object e.g.

```
as.character(ziggy)

## [1] "8.16227766016838"

ziggy

## [1] 8.162278
```

To reassign it we can pass the output of the function and overwrite the original object using the assignment operator `<-` e.g.

```
ziggy <- as.character(ziggy)
ziggy

## [1] "8.16227766016838"
```

We can see that now the object `ziggy` has been changed to a `character` permanently.

You can also create “empty” vectors of specific lengths by using e.g. `numeric()`, `character()`, `logical()` functions. For example, to create an empty vectors of length 10 say, then you can use e.g.

```
y <- numeric(10)
y

## [1] 0 0 0 0 0 0 0 0 0 0

y <- character(10)
y

## [1] "" "" "" "" "" "" "" "" ""

y <- logical(10)
y

## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Note that these are not “empty” specifically, rather they have been filled with identical default values, however these are useful functions for creating objects that you can “fill-in” later on.

Task 4

Use the various vector creation commands to create the following sequences:

1. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11.
2. 2, 4, 6, 8, 10, ..., 50.
3. All of the odd numbers from 17 through to 33.
4. The letters of the days of the week (i.e. M, T, W, T, F, S, S).

Show: Solution on P195

2.4.1.2 Subsetting and re-arranging vectors

If we only want to get at certain elements of a vector then we need to be able to subset the vector. In R, square brackets [] are used to get subsets of a vector.

Important

In R, objects are **indexed** such that the **first** element is element 1 (unlike C or Python for example, which indexes relative to 0). In R the first element of `x` is `x[1]`, whereas in Python it would be `x[0]`.

Note also that in Python the command `x[-1]` would extract the **last** element of the vector `x`, whereas in R the - is used to **remove** elements; hence `x[-1]` would **remove** the **first** element of `x`.

Task 5

Copy the following commands into your script file and then run them in order to see how this works. Annotate each line using comments to describe what it is doing.

```
x <- 1:5
x[3]
x[c(2, 3, 4)]
x[1] <- 17
x[-3]
w <- x[-3]
y <- c(1, 5, 2, 4, 7)
y[-c(3, 5)]
y[c(1, 4, 5)]
i <- 1:3
y[i]
z <- c(9, 10, 11)
y[i] <- z
```

Show: Solution on P196

Task 6

Can you work out what the following functions do?

1. `order(y)`
2. `rev(y)`
3. `sort(y)`

Can you work out a way to sort `y` into decreasing order?

Show: Solution on P197

2.4.1.3 Element-wise (or vectorised) operations

R performs most operations on vectors **element-wise**, meaning that it performs each calculation on each element of a vector or array **independently** of the other elements. This can make for very fast manipulation of large data sets.

Aside

R is a **high-level** language, meaning that it is built on **low-level** languages, in this case C and Fortran. In C, if we wish to apply a function to each element of an array, we need to write a loop. In R there are so-called **vectorised** functions, that will apply a function to each element of a vector or array. We will cover loops in R later on, but generally they are much slower than the equivalent loops in C or Fortran. The source code for vectorised functions are often written in C or Fortran and hence run much faster than writing a loop in native R code. In addition they make the code more concise. For simple functions the difference in run time is often too small to really notice, but for more complex or large-scale functions the difference in run time can be large. To this end, R also offers various ways of using C code within R functions. But that is beyond the scope of this workshop...

Task 7

Copy the following commands into your script file and then run them in order to see how this works. Annotate each line using comments to describe what it is doing.

```
y <- 1:10
y^2
log(y)
exp(y)
x <- c(5, 4, 3, 2, 1, 5, 4, 3, 2, 1)
x + y
x * y
z <- c(-1, 2.2, 10)
z + y
z * y
```

Show: Solution on P197

If you add or multiply vectors of different lengths then R automatically repeats the smaller vector to make it bigger. This generates a warning if the length of the smaller vector is not a divisor of the length of the longer vector. Division and subtraction work in the same way e.g.

```
c(1:6) * c(1:2)

## [1] 1 4 3 8 5 12

c(1:3) * c(1:2)

## Warning in c(1:3) * c(1:2): longer object length is not a multiple of shorter
## object length

## [1] 1 4 3
```

This is a useful property if the shorter vector is of length 1, in which case R multiplies each element of the longer vector by the scalar e.g.

```
c(1:3) * 2

## [1] 2 4 6
```

Important

You must be **very careful** with this. I only use this feature when multiplying a vector by a scalar; otherwise I will always generate vectors of the same length, since I think this makes explicit what you are trying to achieve. Hence instead of writing `c(1:6) * c(1:2)`, I would write `c(1:6) * rep(1:2, 3)`. (I personally would rather that R did not allow elementwise operations on vectors that are of different lengths, unless one of the lengths is one. However, it does, and so one should be careful...)

R will also perform **matrix multiplication** (e.g. dot products) on vectors, and this is described in a [later section](#).

2.4.1.4 Logical vectors and conditional subsetting

Vectors can be created according to logical criteria using the following comparison operators:

Table 2.3: Logical operators

Operator	Definition
<code>!=</code>	not equal
<code><</code>	less than
<code><=</code>	less than or equal to
<code>></code>	greater than
<code>>=</code>	greater than or equal to
<code>==</code>	logical equals
<code> </code>	logical OR
<code>&</code>	logical AND
<code>!</code>	logical NOT

Task 8

Copy the following commands into your script file and then run them in order to see how this works. Annotate each line using comments to describe what it is doing.

```
x <- 1:10
y <- c(5, 4, 3, 2, 1, 5, 4, 3, 2, 1)
x < 4
x[x < 4]
y[x < 4]
y > 1 & y <= 4
y[y > 1 & y <= 4]
z <- y[y != 3]
```

Show: Solution on P198

Finally there are many commands that are specifically aimed at vectors or sets of data. The following is a small selection for reference purposes:

Table 2.4: Common R functions

Function	Description
<code>length(x)</code>	returns the length of vector <code>x</code> (i.e. the number of elements in <code>x</code>)
<code>names(x)</code>	get or set names of <code>x</code> (i.e. we can give names to the individual elements of <code>x</code> as well as just having them numbered)
<code>min(x)</code>	returns the smallest value in <code>x</code>
<code>max(x)</code>	returns the largest value in <code>x</code>
<code>median(x)</code>	returns the median of <code>x</code>
<code>range(x)</code>	returns a vector with the smallest and largest values in <code>x</code>
<code>sum(x)</code>	returns the sum of values in <code>x</code>
<code>mean(x)</code>	returns the mean of values in <code>x</code>
<code>sd(x)</code>	returns the standard deviation of <code>x</code>
<code>var(x)</code>	returns the variance of <code>x</code>
<code>diff(x)</code>	returns a vector with all of the differences between subsequent values of <code>x</code> . This vector is of length 1 less than the length of <code>x</code>
<code>summary(x)</code>	returns different types of summary output depending on the type of variable stored in <code>x</code>

Task 9

Now that we know about **elementwise** operations, try to understand the `nested` function `sum(sqrt(log(seq(1, 10, by = 0.5))))`. To help with this, try to write this out in a series of steps and figure out what is returned at each stage of the nesting, and how this is passed to the next function.

Show: Solution on P199

2.4.1.5 Factors

We have seen that we can create `character` vectors. These are collections of strings, but with no further information. For example, if I try to summarise a character vector, R returns no useful information:

```
x <- c("bob", "sam", "ted")
summary(x)

##      Length   Class    Mode
##      3 character character
```

A `factor` is simply a character vector with some additional “grouping” information attached. For example,

```
x <- factor(c("bob", "sam", "ted"))
x

## [1] bob sam ted
## Levels: bob sam ted

summary(x)
```

```
## bob sam ted
## 1 1 1
```

Now the `x` object has grouped together all elements of the vector that share the same name (called the `levels` of the factor), and the `summary()` function tells us how many entries there are for each level. The `levels()` function will return the levels of a factor:

```
levels(x)
```

```
## [1] "bob" "sam" "ted"
```

R coerces `characters` to factors in **lexicographical order**. If you wish to set specific `levels`, then you can add a `levels` argument to the `factor()` function.

Task 10

Copy the following commands into your script file and then run them in order to see how this works. Annotate each line using comments to describe what it is doing.

```
y <- c(5, 4, 3, 2, 10, 5, 4, 3, 2, 1)
factor(y)
factor(y, levels = 1:10)
y <- as.character(c(5, 4, 3, 2, 10, 5, 4, 3, 2, 1))
factor(y)
factor(y, levels = sort(unique(as.numeric(y))))
y <- c("low", "mid", "mid", "high", "low")
factor(y)
factor(y, levels = c("low", "mid", "high"))
```

Show: Solution on P200

`factor` objects are incredibly useful when working with data, since they essentially code up **categorical** variables, though care must be taken when manipulating them. We explore the use `factor` objects in more detail in a [later section](#) and care must be taken when [converting](#) between factors and other types.

2.4.2 Conversions between object types

Before we progress to other objects, we should have a brief aside to talk about conversion between types. So far we have introduced `numeric` vectors, which store numerical values (note that R makes no distinction between `integer` or `double` types, like C or Python do). We have also seen `character` types, which store strings, and `factor` types that store strings with additional grouping information attached. We have also seen `logical` types, which simply store `TRUE` or `FALSE` values.

It is possible to convert between these different forms, using an `as.TYPE()` function (where `TYPE` is replaced with the correct type of interest). For example, we can easily convert numbers to characters:

```
x <- c(0, 1, 3, 10, 0, 4)
x
```

```
## [1] 0 1 3 10 0 4
```

```
as.character(x)

## [1] "0"  "1"  "3"  "10" "0"  "4"
```

(Notice the double-quotes around the elements of the converted vectors.) They can also easily be converted into factors (if you want to set the levels of the factor specifically, then you can use the `factor()` function as described [earlier](#)):

```
factor(x)

## [1] 0 1 3 10 0 4
## Levels: 0 1 3 4 10
```

We can also convert numbers to logical strings, in which case R converts 0 values to `FALSE` and anything else to `TRUE`. Hence,

```
as.logical(x)

## [1] FALSE TRUE TRUE TRUE FALSE TRUE
```

Conversions of `logical` vectors to `numeric` vectors results in `TRUE` values becoming 1 and `FALSE` values becoming 0.

R can also convert characters to numbers, but only if the conversion makes sense. For example,

```
y <- c("10", "2", "8")
y
```

```
## [1] "10" "2"  "8"
```

```
as.numeric(y)
```

```
## [1] 10  2   8
```

If the conversion doesn't make sense, then R will return a **missing value** (`NA`) and print a warning:

```
y <- c("10", "2", "bob")
as.numeric(y)

## Warning: NAs introduced by coercion

## [1] 10  2  NA
```

Factors are tricky little blighters! Superficially they can look like `numeric` types when printed to the screen, but in fact they are named **groups**, and as such if we convert a `factor` to a `numeric`, then R converts according to the **level** of the factor:

```
y <- factor(c("10", "2", "8"))
y
```

```
## [1] 10 2 8
## Levels: 10 2 8
```

```
as.numeric(y)
```

```
## [1] 1 2 3
```

Notice that this returns 1, 2, 3 rather than 10, 2, 8.

If a `factor` holds a numerical value you wish to convert directly, then you must convert the factor to a `character` first (for which you can often use nested functions to do) e.g.

```
y <- factor(c("10", "2", "8"))
as.numeric(y)
```

```
## [1] 1 2 3
```

```
as.numeric(as.character(y))
```

```
## [1] 10 2 8
```

Conversions to `character` types are straightforward in any case, using `as.character()`. Conversions of `characters` or `factors` to `logical` types will produce missing values (`NA`).

2.4.3 Matrices

Matrices are another way of storing data in R. They can be thought of as 2D versions of vectors; a single structured group of numbers or words arranged in both rows and columns. Each element of the matrix is numbered according to the row and column it is in (starting from the top-left corner). The size of the matrix is given as two numbers which specify the number of rows and columns it has. The size is referred to as the **dimension** of the matrix.

The simplest way of creating a matrix in R is to use the `matrix()` function.

Task 11

Copy the following commands into your script file and then run them in order to see how this works. Annotate each line using comments to describe what it is doing.

```
matrix(1:4, 2, 2)
matrix(1:4, 2, 2, byrow = TRUE)
matrix(0, 3, 4)
```

Show: Solution on P200

Like vectors, matrices can also contain `character` values e.g.

```
matrix(letters[1:9], 3, 3)
```

```
##      [,1] [,2] [,3]
## [1,] "a"  "d"  "g"
## [2,] "b"  "e"  "h"
## [3,] "c"  "f"  "i"
```

Note

All elements of a `matrix` must be of the same type. You can have a `numeric` matrix, or a `character` matrix, but not a mixture. If you try to specify a mixture then R will convert all entries to the most complex type, which can lead to unexpected consequences unless you are careful!

2.4.3.1 Subsetting matrices

Getting access to elements of a matrix follows on naturally from getting access to elements of a vector. In the case of a matrix though you must specify both the row and column of the element that you want, using the format `[ROW, COL]`.

Task 12

Copy the following commands into your script file and then run them in order to see how this works. Annotate each line using comments to describe what it is doing.

```
vals <- c(1, 2, 3, 4, 5, 0.5, 2, 6, 0, 1, 1, 0)
mat <- matrix(vals, 4, 3)
mat[2, 3]
mat[1, ]
mat[, 3]
mat[-2, ]
mat[c(1, 3), c(2, 3)]
```

Show: Solution on P201

2.4.3.2 Creating matrices

As well as using the `matrix()` function, we can also create matrices by binding together several vectors (or other matrices). We use two commands here: `cbind()` and `rbind()`.

Task 13

Copy the following commands into your script file and then run them in order to see how this works. Annotate each line using comments to describe what it is doing.

```
x1 <- 1:3
x2 <- c(7, 5, 6)
x3 <- c(12, 19, 25)
cbind(x1, x2, x3)
rbind(x1, x2, x3)
```

Show: Solution on P201

Some useful commands for matrices are listed in Table 2.5.

Table 2.5: Useful `matrix` commands in R

Function	Example	Description
<code>matrix()</code>	<code>matrix(0, 3, 4)</code>	Creates a (3×4) matrix filled with zeros
<code>dim()</code>	<code>dim(mat)</code>	Returns the dimensions of a matrix <code>mat</code> in the form (rows \times columns)
<code>t()</code>	<code>t(mat)</code>	Transpose a matrix <code>mat</code>
<code>rownames()</code>	<code>rownames(mat)</code>	Returns or sets a vector of row names for a matrix <code>mat</code>
<code>colnames()</code>	<code>colnames(mat)</code>	Returns or sets a vector of column names for a matrix <code>mat</code>
<code>cbind()</code>	<code>cbind(v1, v2)</code>	Binds vectors or matrices together by column
<code>rbind()</code>	<code>rbind(v1, v2)</code>	Binds vectors or matrices together by row

2.4.3.3 Elementwise operations

Elementwise operations also work similarly to vectors, though even more care must be taken when the vectors have different lengths.

Task 14

Copy the following commands into your script file and then run them in order to see how this works.
Annotate each line using comments to describe what it is doing.

```
x <- matrix(1:9, 3, 3)
x
x * 2
x * c(1:3)
```

Show: Solution on P202

2.4.3.4 Matrix multiplication

Another key aspect of matrix manipulation that is required for many algorithms is **matrix multiplication**. R does this by using the `%*%` syntax. Note the differences between **matrix multiplication** and **elementwise multiplication**:

```
x <- matrix(1:9, 3, 3)

## matrix multiplication
x %*% x
```

```
##      [,1] [,2] [,3]
## [1,]    30   66  102
## [2,]    36   81  126
## [3,]    42   96  150
```

```
## elementwise multiplication
x * x
```

```
## [,1] [,2] [,3]
## [1,]    1   16   49
## [2,]    4   25   64
## [3,]    9   36   81
```

Matrix multiplication also works with vectors. In this case the vector will be promoted to either a row or column matrix to make the two arguments conformable. If two vectors are of the same length, it will return the inner (or dot) product (as a matrix). Compare the following (taking note of the comments):

```
## setup a vector of length 3
x <- 1:3

## conduct elementwise multiplication
x * x
```

```
## [1] 1 4 9
```

```
## return matrix multiplication of (1 x 3) and (3 x 1) matrix
t(x) %*% x
```

```
## [,1]
## [1,] 14
```

```
## matrix multiplication of (3 x 1) and (1 x 3) matrix
x %*% t(x)
```

```
## [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    2    4    6
## [3,]    3    6    9
```

```
## create new vector
y <- 5:6
```

```
## return matrix multiplication of (3 x 1) and (1 x 2) matrix
x %*% t(y)
```

```
## [,1] [,2]
## [1,]    5    6
## [2,]   10   12
## [3,]   15   18
```

```
## set up some new matrices
x <- matrix(1:6, 3, 2)
y <- matrix(1:8, 2, 4)
```

```
## return matrix multiplication of (3 x 2) and (2 x 4) matrix
x %*% y
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    9   19   29   39
## [2,]   12   26   40   54
## [3,]   15   33   51   69
```

2.4.4 Lists

An R list is an object consisting of an ordered collection of objects known as its components (or elements). There is no particular need for the components to be of the same type (unlike vectors), and, for example, a list could consist of a numeric vector, a logical value, a matrix, a complex vector, a character array, and so on.

The following command creates a list object called `simpsons`, with four components: the first two are character strings, the third is an integer, and the fourth component is a numerical vector. Try creating a list as follows:

```
simpsons <- list(
  father = "Homer",
  mother = "Marge",
  no_children = 3,
  child_ages = c(10, 8, 1)
)
```

The components of a list are always numbered and may always be referred to as such. Thus with `simpsons` above, its components may be individually referred to as `simpsons[[1]]`, `simpsons[[2]]`, `simpsons[[3]]` and `simpsons[[4]]`. Since `simpsons[[4]]` is a **vector**, we can access the elements of the vector using the `[]` notation (i.e. `simpsons[[4]][1]` is its first element).

The command `length(simpsons)` gives the number of components it has, in this case four.

The components of lists may also be named (here the names are `father`, `mother`, `no_children` and `child_ages`) and in this case the component may be referred to either by giving the component name as a character string in place of the number in double square brackets i.e. `simpsons[["father"]]` (note the double-quotes), or, more conveniently, by giving an expression of the form `simpsons$father` (note no double-quotes, since `father` is an object contained within `simpsons`).

This means that the following will all return the first component of the list `simpsons`:

```
simpsons[[1]]
```

```
## [1] "Homer"
```

```
simpsons[["father"]]
```

```
## [1] "Homer"
```

```
simpsons$father
```

```
## [1] "Homer"
```

The command `names(simpsons)` will return a vector of the component names of the list `simpsons`:

```
names(simpsons)

## [1] "father"      "mother"       "no_children"  "child_ages"
```

Note the strange double square bracket notation. We can also subset a list using single square brackets, though the returned object is different. For example, typing

```
simpsons[1]

## $father
## [1] "Homer"

is.list(simpsons[1])

## [1] TRUE
```

returns a `list` object, of length 1, where the first element of the list is called `father` and is a `character` vector. Using double square brackets returns the `character` vector itself i.e.

```
simpsons[[1]]

## [1] "Homer"

is.list(simpsons[[1]])

## [1] FALSE
```

Sometimes we might wish to extract certain components of a list, but keep a list structure, for example

```
simpsons[c(1, 3)]

## $father
## [1] "Homer"
##
## $no_children
## [1] 3
```

produces a `list` of length 2, containing two components: the first is a `character` vector called `father`, and the second is a `numeric` vector called `no_children`.

Lists can be useful ways of collecting together a wide range of different bits of information that are all related. R naturally uses lists in a variety of situations but most commonly they are produced as a result of a function call. (In fact most custom classes in R are essentially lists, but with specific methods associated with that class e.g. `plot` or `summary` functions for example).

2.4.5 Data frames

A `data.frame` object is a special type of list where all of the components are in some sense the same size. The simplest form of `data.frame` consists of a list of vectors all of the same length, but not necessarily of the same type (`numeric`, `character` or `logical`). In this case a `data.frame` object may be regarded as a matrix with columns for each vector. It can be displayed in a matrix layout, and the data in its rows and columns extracted using the matrix subsetting notation (`[ROWS, COLS]`). (However, because it is also a special type of `list`, we can also extract columns by using the `DFNAME$COLNAME` notation, where `DFNAME` is the name of the `data.frame` of interest, and `COLNAME` is the name of a specific column. The `[[[]]]` notation also works for `data.frame` objects.)

The important difference (to R at least) between a `matrix` object and a `data.frame` object that only contains `numeric` vectors, is that R thinks of the matrix as a single object, but it thinks of the `data.frame` as several connected, but distinct objects. (To the user it can be hard to distinguish between objects, and the commands `is.matrix()`, `is.list()` and `is.data.frame()` will return `TRUE` or `FALSE` depending on whether an object is a `matrix`, a `list` or `data.frame` respectively.)

Note

`data.frames` are very important for statistical analysis, as they frequently fit the form of available data. They are so important in fact that a lot of statistical functions actually expect this.

2.5 Reading in data

Most data are stored in e.g. Excel files. Although some packages allow Excel files to be read directly into R, it is (in my opinion) much better practice to save the data as a **text file**, since this strips out unnecessary formatting and results in smaller file that can be universally used by any end user. Excel also has lots of other problems; see e.g. [here](#) for various examples.

Aside

You can save Excel spreadsheets as comma-separated text files using e.g. *File > Save As > ...*, then from the ‘Save as type’ options choose ‘CSV (comma separated)’. There is no need to add a suffix, because Excel will automatically add `.csv` to your file name.

`.csv` files can be read into R directly as `data.frame` objects, using the `read.csv()` function.

A `.csv` file is simply a text file where each element is separated by a comma (the delimiter). This can be easily viewed and manipulated in any text editor (such as Notepad). Many R users prefer to use ‘Text – Tab delimited’ data files. In this case each entry is delimited by a tab. In fact, one can use any delimiter one prefers, but I often prefer commas, which are so common they have their own suffix! If you choose the tab-delimited route, you have to be **extra careful** about spaces between words in your spreadsheet, and to read the data file into R you should use `read.table()` instead of `read.csv()` (where you can explicitly specify the delimiter as an argument to the function). This is similarly true if using `.csv` files: one must be careful not to include entries with commas in them.

Note: `read.csv()` might fail if there are any spaces between words within data cells. If this happens, replace all these spaces by underscores `_` before saving the spreadsheet in Excel.

In fact, it is good practice to store your data in a much simpler way than many people are used to. Generally, try to avoid spaces between words at all in your data, even in column headers. Underscores are much safer. Try to avoid using long words (since you will have to type them out each time you want to access the corresponding elements), and often I will try to convert as much to lower case as possible (since R is case sensitive, so it speeds up your typing).

2.5.1 Example: sexual reproduction in fruitflies

These are data from [Partridge and Farquhar \(1981\)](#), when at the time the paper was published the cost of increased reproduction in terms of reduced longevity had been shown for female fruitflies, but not for males. The authors set up an experiment that used a factorial design to assess whether increased sexual activity affected the lifespan of male fruitflies.

The flies used were an outbred stock. Sexual activity was manipulated by supplying individual males with one or eight receptive virgin females per day. The longevity of these males was compared with that of two control types. The first control consisted of two sets of individual males kept with one or eight newly inseminated females. Newly inseminated females will not usually remate for at least two days, and thus served as a control for any effect of competition with the male for food or space. The second control was a set of individual males kept with no females. There were 25 males in each of the five groups, which were treated identically in number of anaesthetisations (using CO₂) and provision of fresh food medium.

The data should have the following columns:

- **partners**: number of companions (0, 1 or 8).
- **partner type**: type of companion (inseminated female; virgin female; control (when ‘partners = 0’)).
- **longevity**: lifespan, in days.
- **thorax**: length of thorax, in mm.

The file `ff.csv` contains the data from this experiment. Download this data set to your working directory. (You can view this file in a text editor if you so wish.) We can load this into R using the `read.csv()` function, here storing the resulting `data.frame` as an R object called `ff`:

```
ff <- read.csv("ff.csv")
```

Note

If R can't find the file, check it's in the working directory, or pass the full path to `read.csv()`.

```
head(ff)
```

```
##   partners partner.type longevity thorax
## 1       8 Inseminated      35    0.64
## 2       8 Inseminated      37    0.68
## 3       8 Inseminated      49    0.68
## 4       8 Inseminated      46    0.72
## 5       8 Inseminated      63    0.72
## 6       8 Inseminated      39    0.76
```

```
summary(ff)
```

```
##      partners   partner.type      longevity      thorax
## Min.   :0.0   Length:125   Min.   :16.00   Min.   :0.640
## 1st Qu.:1.0  Class  :character 1st Qu.:46.00  1st Qu.:0.760
## Median  :1.0  Mode   :character Median  :58.00  Median  :0.840
## Mean    :3.6                    Mean   :57.44  Mean   :0.821
## 3rd Qu.:8.0                    3rd Qu.:70.00 3rd Qu.:0.880
## Max.    :8.0                    Max.   :97.00  Max.   :0.940
```

Looking at the data we can see a number of things:

- Since the original data had spaces in the column names, these spaces have been replaced by dots.
- Columns that can be converted into numbers in sensible ways are stored as such (e.g. the `thorax` column). You can tell this because the `summary()` function returns numerical summaries.
- Columns that can't be easily converted are stored as `character` columns (e.g. the `type` column). You can tell this because the `summary()` function returns no useful summaries for these columns except to say they are `character` columns¹.

These defaults can be useful sometimes for finding errors. For example, if a column of numbers contains a typo that prevents an entry from being converted to a number, then the whole column will be read in as a `character()`. Thus if you spot this you can often track down the typo and correct it. Please see the section on [converting between types](#) for further details on how to manually convert between types.

Important

It is a good idea to always check and read-in data to ensure each column is in the correct format before progressing any further with your analysis. A common error is that often categorical variables are coded as numbers. For example, a data set might contain numbers 0 or 1 to denote e.g. died/survived. In this case R will read the column in as a `numeric`, when it should be a `factor`. In this case you would want to manually convert your column to a `factor` before proceeding.

Task 15

Convert the `partners` column of the `ff` data frame into a `factor`. Do the same for `partner.type`. Produce a summary on the revised `ff` object. [Hint: you can do this by overwriting the relevant columns in `ff`.] What do you notice about the summary?

Show: Solution on P202

Objects saved as an [.rds file](#) can be read in and assigned to objects using the `readRDS()` function e.g.

```
## save worms data frame as an .rds file
saveRDS(ff, "ff.rds")

## read in as a new object
ff1 <- readRDS("ff.rds")
```

Here the new `ff1` object is identical to the original `ff` object. I often read in `.csv` files, clean them up into the correct formats, and then save as an `.rds` file for further analysis. Since the `.rds` file stores R objects directly, then reading in the `.rds` file means I don't have to do all the cleaning up again. For large data sets, the resulting `.rds` file can often be a much smaller file size than the original text file.

Alternative approaches can be found in other packages, and indeed a good package is `readr`, which is part of the `tidyverse` suite of packages—see [here](#) for a useful cheat sheet. Some other options can be found [here](#).

Once we have data read in as a `data.frame` object, we can do all sorts of useful things with them, so-much-so that most of the rest of the module will be built around using these objects. As a little example, consider the ease with which we can produce a mean thorax length for control flies with just a short piece of code:

¹note that in versions of R before R 4.0.0 these columns would have been automatically converted to `factor` columns. Whilst this is often a useful thing to do, it is much better now that R does not do this automatically and instead forces you to do this explicitly in your code so that it is documented.

```
## calculate mean thorax length for control flies
mean(ff[ff$partner.type == "Control"])
```

```
## [1] 0.836
```

We will see many more complex examples in coming practicals.

2.6 Saving outputs

R allows you to save various things, with regards to inputs we can save:

- the full history of all commands used in the current session—via `savehistory()`. This creates text files with ‘.Rhistory’ suffixes.
- Individual script files (as text files with ‘.R’ suffixes). We’ve already seen these.

With regards to outputs we can save:

- the current R workspace—via `save.image()`. This creates a **binary** file with a ‘.RData’ suffix. This can be reloaded into R using the `load()` function, and includes all objects in the current workspace.
- A selection of objects—via the `save()` function. Here we can extract subsets of objects and save them. For example, `save(x, y, file = "practical1.RData")` will save only the objects `x` and `y` into a file called “practical1.RData”. Alternatively a character vector of objects can be supplied e.g. `save(list = c("x", "y"), file = "practical1.RData")` will do the same thing as the previous command.
- A single object can be saved using `saveRDS()`. This saves into a binary file with a suffix ‘.rds’. This is really useful for saving single objects, often cleaned `data.frame` objects, which can be loaded into R without having to recreate from scratch again. For example, `saveRDS(x, "x.rds")` would save the `x` object into a file called “`x.rds`”. This can be loaded into a new session via the `readRDS()` function. Another useful feature is that `readRDS()` can be used to pass the contents of a .rds file directly into an object of your choice e.g. `y <- readRDS("x.rds")` will create a new object called `y`, that contains the contents of the file “`x.rds`”.

Part II

Programming

Chapter 3

Programming—Practical 1

Nik Cunniffe (njc1001@cam.ac.uk)

3.1 Getting started

Start a new RStudio session, set your current working directory, open a new R Script and save this (blank) script to a file with an appropriate name.

Use this script file to edit and keep copies of the code you write in this practical: you may find it useful be able to look back later.

3.2 Predicting the behaviour of for loops

We will use R to generate some data in the form of vectors and use these data to form some looping constructs. Predicting what code will do and testing your predictions by running the code should help cement your understanding of for loops.

Type the following code into your script. Remember that code after # is a comment and will be ignored by the R interpreter (you do not have to type these comments in for yourself, but you might want to if you think you will need to refer to them later).

```
a <- 1:10          # Assign the integers 1 to 10 in order to the vector a
print(a)           # Print a to the screen
b <- sample(a)    # Set b to be a with elements shuffled into random order
print(b)           # Print b to the screen
```

Run the code a few times either by sourcing it or by highlighting it in the script then pressing the run button in RStudio.

You should note that:

- The first line creates a new vector **a** with the elements 1 to 10 in order.
- The second line prints the vector **a** to the screen.
- The third line uses a built-in R function to sample (all of) the elements of the vector **a** without replacement (i.e. to permute the elements into a random order), and assigns the result to the vector **b**.
- The fourth line prints the vector **b**.

You should also note that the code gives a different result each time you run it, since repeating the `sample` command leads to `b` being set to a new permutation of `a`.

Task 16

Write down what you expect the output of the following code to be before verifying your answer by entering the code into your script and running it.

```
for (i in 1:10) {  
    print(paste(i, a[i]))  
}
```

Show: Solution on P202

Task 17

Write down what you expect the output of the following code to be before verifying your answer by entering the code into your script and running it.

```
for (i in 10:1) {  
    print(paste(i, b[i]))  
}
```

Show: Solution on P203

Task 18

Write down what you expect the output of the following code to be before verifying your answer by entering the code into your script and running it.

```
for (i in a) {  
    print(paste(i, b[i]))  
}
```

Show: Solution on P203

Task 19

Write down what you expect the output of the following code to be before verifying your answer by entering the code into your script and running it.

```
for (i in sample(a)) {  
    print(paste(i, b[i]))  
}
```

Show: Solution on P203

3.3 Writing a for loop

Adapt the code from the lecture to use a `for` loop to do the following:

Task 20

Print the 12 times table using a `for` loop, i.e. produce output of the form

- 1 times 12 is 12
- 2 times 12 is 24
- ...
- 12 times 12 is 144

Show: Solution on P204

3.4 Writing a while loop

Repeat the previous task using a `while` loop.

Task 21

Print the 12 times table using a `while` loop, i.e. produce output of the form

- 1 times 12 is 12
- 2 times 12 is 24
- ...
- 12 times 12 is 144

Show: Solution on P204

3.5 Using loops to do more complex calculations

3.5.1 A loop for which the number of iterations is known in advance

Enter the following code to your script and check you understand the output.

```
p <- 4
print(2^p)
p <- 5
print(p^3)
```

Now write code to answer the following question.

Task 22

What is the sum of the first 15 powers of 2 (i.e. $2^1 + 2^2 + \dots + 2^{15}$)?

Show: Solution on P205

3.5.2 A loop for which the number of iterations is not known in advance

Write code to answer the following question.

Task 23

What is the smallest power of 2 that is greater than 1,000,000?

Show: Solution on P205

3.6 Writing simple functions

Type the following code into your script and run it. Before you run it, think about whether you expect any output.

```
sayHello <- function() {  
  print("Hello")  
}
```

Type the following at the command prompt, but before you do so, predict what will happen.

```
sayHello()
```

Now type the following code into your script.

```
sayHelloWithArg <- function(whoTo) {  
  print(paste("Hello", whoTo))  
}
```

Task 24

Now type each of the following instructions into the command prompt. Before typing each line, predict what will happen.

```
sayHelloWithArg()  
sayHelloWithArg  
sayHelloWithArg("nik")  
sayHello("nik")  
sayHelloWithArg(nik)  
sayHelloWithArg(whoTo="nik")  
sayHelloWithArg(whoTo="nik")
```

Show: Solution on P205

3.7 Predicting the behaviour of a function

For now, do not type in the following code, but instead just read it.

```
cat <- 2
canary <- 4
buster <- function(cat, canary) {
  cat <- cat*2
  canary <- canary*3
  return(cat + canary)
}
ghost <- buster(canary, -cat)
```

Task 25

Think carefully about what you expect the values of `cat`, `canary` and `ghost` to be after this code is executed and write down your answer. Now execute the code and test your prediction.

Show: Solution on P206

3.8 Return values from functions

The code in the box below does the following.

- Defines a function `mySum` that returns the sum of a vector.
- Creates a vector `x` containing 100 uniform random numbers between 0 and 1.
- Finds their sum using `mySum`.
- Finds their sum using the built-in R function `sum`.
- Prints out a comparison of the two values.

Type the code into your script and make sure you are happy with how it works.

```
mySum <- function(toSum) { # my function to sum elements of a vector
  retVal <- 0
  for(i in 1:length(toSum)) {
    retVal <- retVal + toSum[i]
  }
  return(retVal)
}

a <- runif(100)      # generate 100 uniform random numbers
myWay <- mySum(a)    # calculate sum using our mySum() function
rWay <- sum(a)       # calculate sum using built-in sum() function
print(paste("My answer", myWay, "... R answer", rWay))
```

Task 26

Use the function `mySum` as a base to write a new function `mySumAndMean`.

The new function should calculate the sum and the mean of whatever vector it receives as an input. It should return a vector: the first element of the output vector should be the sum, the second the mean. You might find the built-in function `mean` helpful to test your code.

Show: Solution on P206

3.9 Additional tasks for those with programming experience

The following tasks are mainly aimed at those of you who already have experience in programming.

3.9.1 Extended example one: Monte Carlo simulation

[Note this task is optional, and you should do it only if you have time]

When I was younger, I sometimes collected Panini football stickers.

The basic idea is very simple: you collect numbered stickers to stick into an album, continuing to buy stickers until you have the full set. The problem is that you don't know which stickers you are going to get until you open a packet. Each packet contains a random sample of the available stickers, and packets are sealed at the time of purchase. This means that as you get closer and closer to completing the album, it becomes more and more unlikely that any new packet will contain any stickers you need for your album.

At the time, it was a matter of significant regret for me that I never did complete the album for the Mexico '86 World Cup. That album required a total of 427 stickers.

Now, better trained in mathematics and computing, I wondered how many stickers I might have had to buy to be successful in my quest to complete the album?

We are going to write a simple “Monte Carlo” simulation to understand this. Monte Carlo simulations rely on random sampling to obtain an estimate of a random variable. You will be writing Monte Carlo simulations of stochastic epidemic models later in the course, so this is good practice.

We remove some of the complexity by considering a slightly simpler situation, in which stickers come in packets of one.

Note the built-in R function `sample` that was introduced earlier can in fact be called in other ways. Type the following into the command line and run it a few times

```
sample(1:200,3)
```

You should see that—when it is called with two arguments—the built-in R function `sample` no longer merely shuffles the whole of the vector that is its first argument, but instead randomly samples the number of elements given by the second argument from that vector (again without replacement). The above corresponds to randomly choosing a three numbers from the list of integers between 1 and 200.

Furthermore, note the behaviour of the built-in R function `rep` by typing the following in at the command line

```
rep(TRUE, 4)
```

You should see that this line of code creates a vector of length 4 where all four elements are the logical value TRUE.

One way of writing a simulation of a single attempt to collect the entire album would be something like the code that follows. **However, this code contains a deliberate mistake.**

Task 27

Enter the code into your script, find the mistake, fix it, and run the code.

```
totStickers <- 427
gotSticker <- rep(FALSE, totStickers)
numLeft <- totStickers
numStickersBought <- 0
while(numLeft > 0) {
  numStickersBought <- numStickersBought + 1
  thisSticker <- sample(1:totStickers, 1)
  if(gotSticker[thisSticker] == TRUE) {
    gotSticker[thisSticker] <- TRUE
    numLeft <- numLeft - 1
    print(paste('got', thisSticker, 'still need', numLeft))
  }
}
print(paste('bought', numStickersBought))
```

Show: Solution on P206

Each time you run the fixed version of the code should lead to a different result. You can get an estimate of the average number of stickers that is required to complete the album by running the code lots of times, and averaging the result.

Task 28

Add a for loop around the code above to run 100 simulations of collecting an entire album, storing the number of stickers required in each simulation in a vector. Apply the function built-in R function `mean` to the vector to estimate the average number of stickers that must be bought to complete the album.

Show: Solution on P207

The expected value (i.e. mean) number of stickers that is required to complete an album of 427 stickers is in fact $427 \times H_{427}$, where H_{427} is the 427th Harmonic number (the mathematics of this is explained in the info box below if you are interested).

It turns out that $H_{427} \approx 6.635$, and so the estimated mean is about 2,800 or so. This is a lot of stickers! The requisite number would be reduced in practice, because you are able swap duplicate stickers with your friends, as well as send off to Panini on a special form for a maximum of 50 stickers towards the end, but nevertheless, this simulation has explained my disappointment as a 9-year-old child!

Show: Relating sticker collecting to the harmonic numbers on P265

3.9.2 Extended example two: the logistic map

[Note this task is optional, and you should do it only if you have time]

The logistic map is a simple population model in discrete-time, and relates the size of a population in generation $n + 1$ to its size in generation n . It is one of the simplest models that shows “chaotic” behaviour.

One way of writing the logistic map is via the following recurrence

$$x_{n+1} = rx_n(1 - x_n)$$

where x_n is the population size in year n , measured as a fraction of the carrying capacity, and r is the intrinsic rate of increase (a growth rate).

Task 29

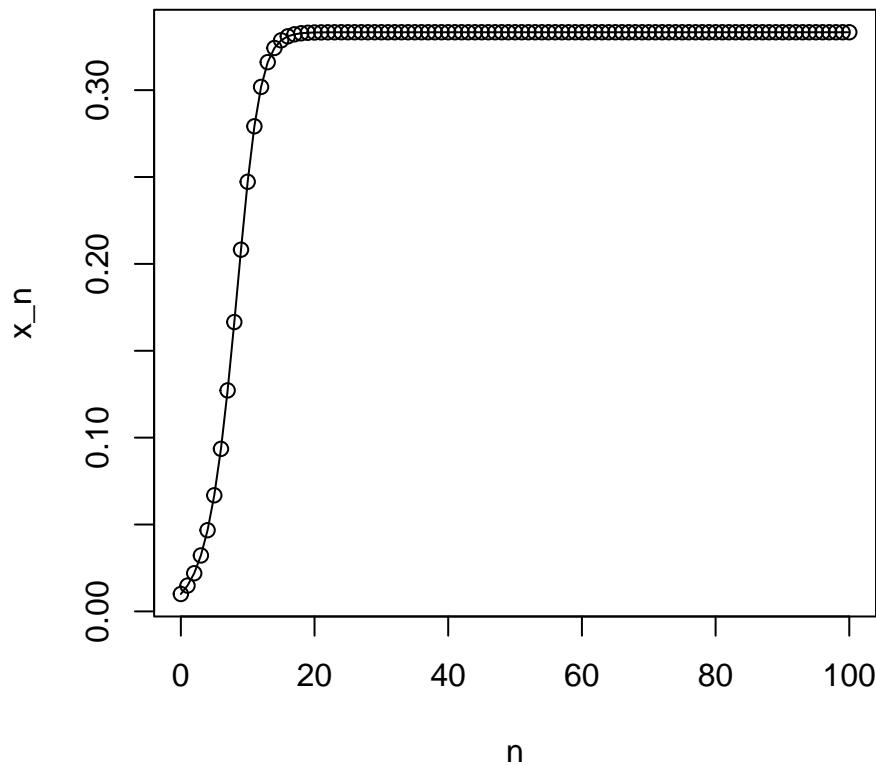
What do you think is the purpose of the following function?

```
logisticValues <- function(r, N, x0) {
  retVal <- numeric(N)
  thisX <- x0
  for(i in 1:N) {
    thisX <- r * thisX * (1 - thisX)
    retVal[i] <- thisX
  }
  return(retVal)
}
```

Show: Solution on P208

Copy the function `logisticValues` into your script file. Also include the following code that uses the function to plot a graph.

```
r <- 1.5      # growth rate
N <- 100      # max generation
x0 <- 0.01    # initial value
xn <- logisticValues(r, N, x0)  # next N starting at x0
plot(0:N, c(x0, xn), type = 'o', xlab = 'n', ylab = 'x_n')  # plot
```



This code shows the behaviour of the logistic map with $r = 1.5$ for 100 generations. Note the slightly clunky way of making sure the initial point is returned.

Task 30

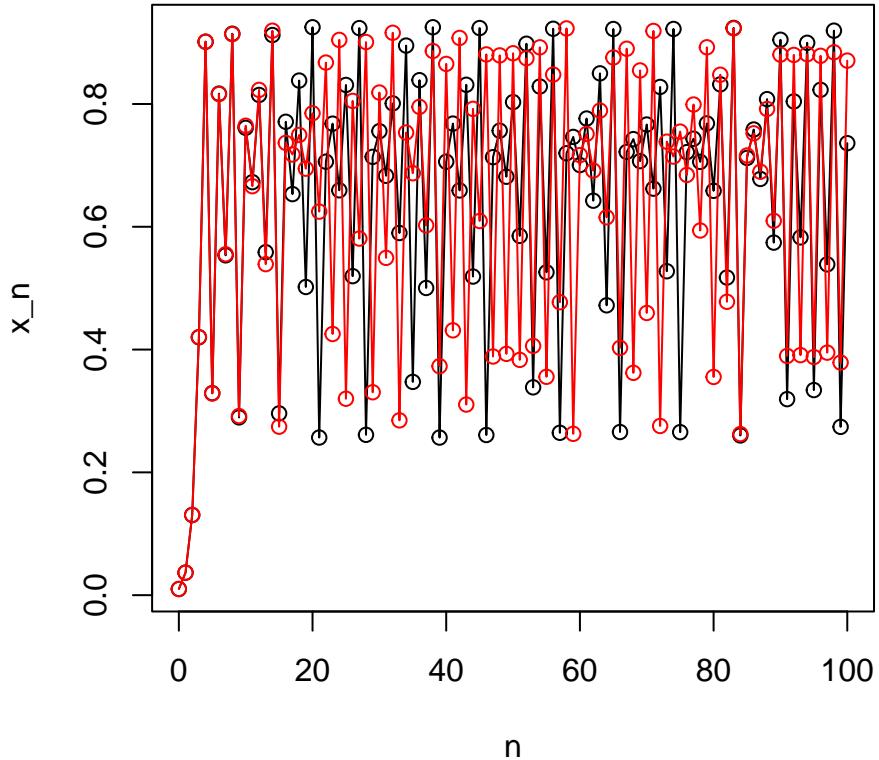
Try running the code for different values of r . What does the graph show you?

Good values of r to try would be $r = 0.5$, $r = 1.5$, $r = 2.5$, $r = 3.1$, $r = 3.5$, $r = 3.7$. You should also test the behaviour for different initial conditions: suitable values would be $x_0 = 0.01$ and $x_0 = 0.1$.

Show: Solution on P208

You should note that in the case $r = 3.7$, the pattern of successive population sizes appears to be fairly random. It also depends on the initial value of x_0 . The following code confirms this (note the first five lines are almost exactly as they were before): include this code in your script to check this.

```
r <- 3.7          # growth rate
N <- 100         # max generation
x0 <- 0.01        # initial value
xn <- logisticValues(r, N, x0)    # next N starting at x0
plot(0:N, c(x0, xn), type = 'o', xlab = 'n', ylab = 'x_n') # plot
x0 <- 0.01001      # different initial value
xn <- logisticValues(r, N, x0)    # next N starting there
lines(0:N, c(x0, xn), type = 'o', col = 'red') # overlay on plot
```



Notice that, despite starting very close together, the trajectories diverge over time. “Sensitive dependence on initial conditions” is characteristic of deterministic chaos.

Task 31

What do you think the following code does?

Test your understanding by typing the function in and running it by issuing an appropriate call to the function at the command prompt.

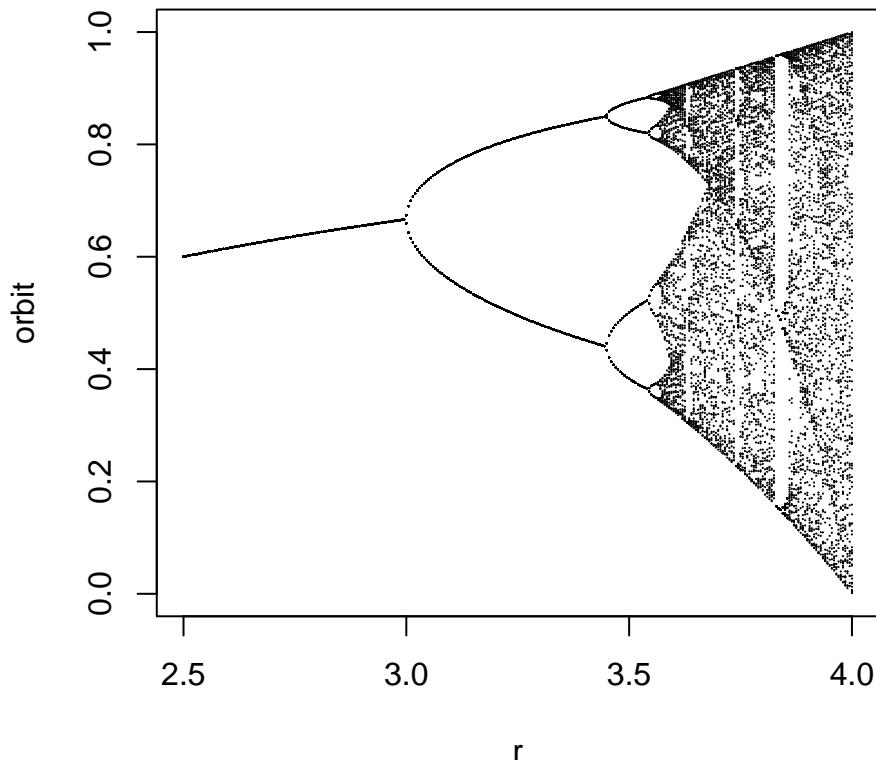
```
logisticValuesAfterBurnIn <- function(r, N, M, x0) {
  retVal <- numeric(M)
  thisX <- x0
  for(i in 1:N) {
    thisX <- r * thisX * (1 - thisX)
  }
  for(i in 1:M) {
    thisX <- r * thisX * (1 - thisX)
    retVal[i] <- thisX
  }
  return(retVal)
}
```

Show: Solution on P208

Enter the following code and run it.

```
rVals <- seq(2.5, 4, by = 0.005)
plot(c(2.5, 4), c(0, 1), type = 'n', xlab = 'r', ylab = 'orbit')
```

```
for(r in rVals) {  
  N <- 1000  
  M <- 100  
  xn <- logisticValuesAfterBurnIn(r, N, M, 0.01)  
  points(rep(r, M), xn, pch = 19, cex = 0.01)  
}
```

**Question 32**

Do you understand what the plot—a so-called “bifurcation diagram”—is showing?

Show: Answer on P208

A very readable introduction to deterministic chaos in biology is given by [May \(1976\)](#). That article is widely available online and comes highly recommended.

Chapter 4

Programming—Practical 2

Matt Castle (mdc31@cam.ac.uk)

4.1 Random variables

A random variable can be thought of as the outcome of an experiment or a measurement i.e. something that can change each time you look at it.

Examples:

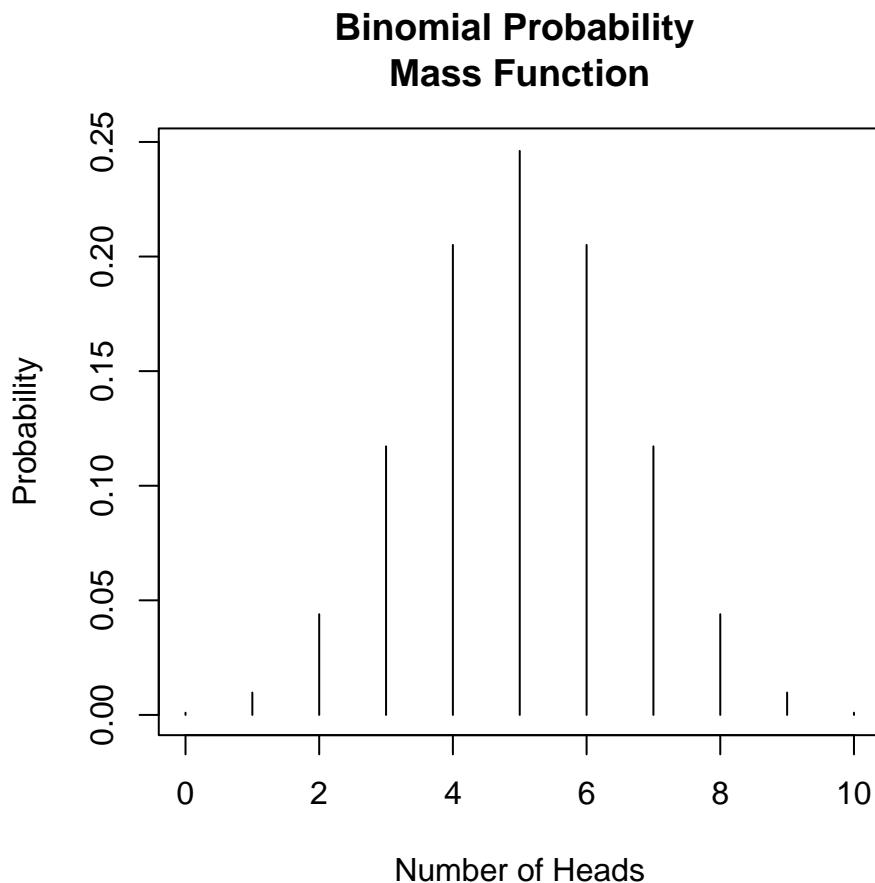
- The number of heads you obtain when tossing a coin 10 times
- The number of emails you received in one hour
- The lifetime of a light-bulb
- The length of time an individual is infected for before recovering

4.2 Probability distributions

A probability distribution will tell us the probability of a random variable taking a specific value.

A binomial distribution will be able to give us the probability of obtaining x heads if we toss a coin 10 times (where x is any number).

```
plot(  
  x = 0:10,  
  y = dbinom(0:10, 10, prob = 0.5),  
  type = "h",  
  main = "Binomial Probability \nMass Function",  
  xlab = "Number of Heads",  
  ylab = "Probability"  
)
```

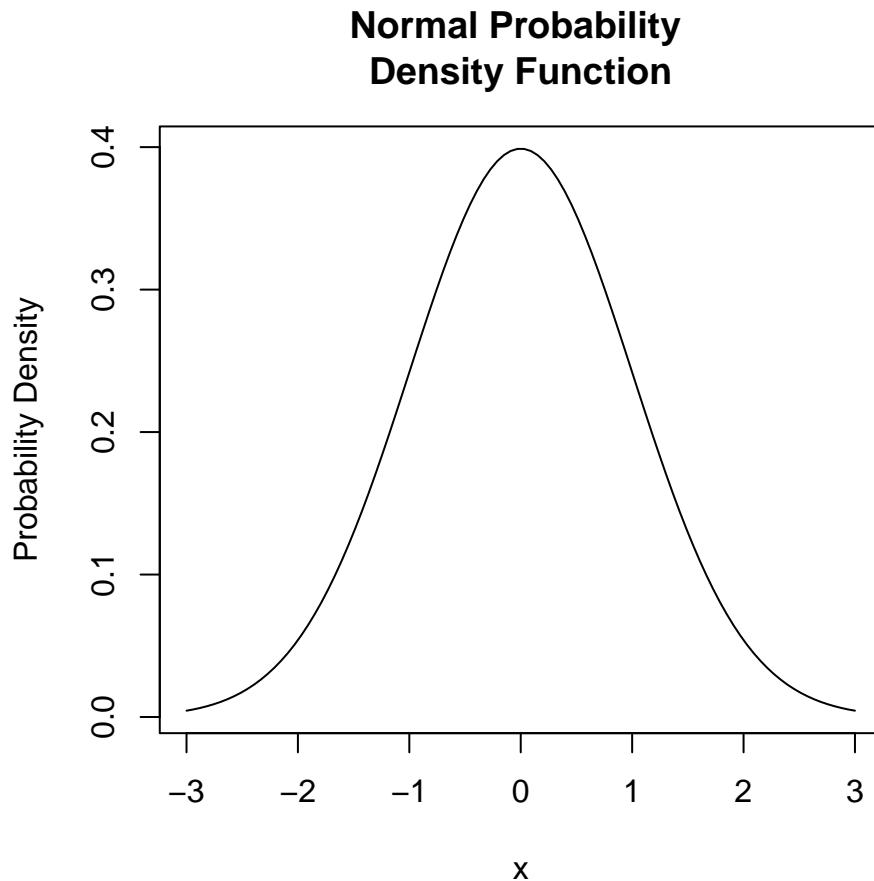


This is an example of a discrete random variables, and so the distribution is called **probability mass function**.

For continuous random variables we can only calculate the probability of being in an interval.

A normal distribution gives us the probability of measurement errors, x , being a certain size.

```
plot(
  x = seq(-3, 3, length = 100),
  y = dnorm(seq(-3, 3, length = 100), mean = 0, sd = 1),
  type = "l",
  main = "Normal Probability \nDensity Function",
  xlab = "x",
  ylab = "Probability Density"
)
```



This is an example of a continuous random variable, and so the distribution is called a **probability density function**.

4.3 Using probability distributions in R—types

R has a large number of inbuilt probability distributions.

Discrete distributions:

- Binomial
- Geometric
- Poisson
- Negative Binomial
- Hypergeometric
- Multinomial

Continuous distributions:

- Normal
- Uniform
- T (T-test)
- F (F-test)
- Exponential

- Cauchy
- Beta
- Gamma
- Weibull
- Chi-squared
- Logistic

R deals with them all in essentially the same way. Each inbuilt distribution in R is referred to by its abbreviation.

- Binomial: `binom`
- Poisson: `pois`
- Geometric: `geom`
- Normal: `norm`
- Uniform: `unif`
- Exponential: `exp`
- ...

Each inbuilt distribution in R has the same four generic functions associated with it.

For a distribution with abbreviation `dist`:

- `ddist()`: density/mass function
- `pdist()`: cumulative density/mass function
- `rdist()`: random number generator function
- `qdist()`: quantile function

4.3.1 `ddist()`

For a distribution type `dist`, the function `ddist(x, ...)` returns:

- The probability $P(X = x)$ (for discrete variables) or
- The probability density at x (for continuous variables)

This can be used to calculate actual probabilities for discrete distributions.

4.3.1.1 `dbinom()`—discrete distribution

Consider a binomial distribution with 10 trials, each with a probability of success of 0.5. The R function `dbinom()` describes the distribution of the probability of success.

It takes the following arguments:

- `x`: the number of successful trials
- `size`: the total number of trials
- `prob`: the probability of success

```
dbinom(4, 10, 0.5)
```

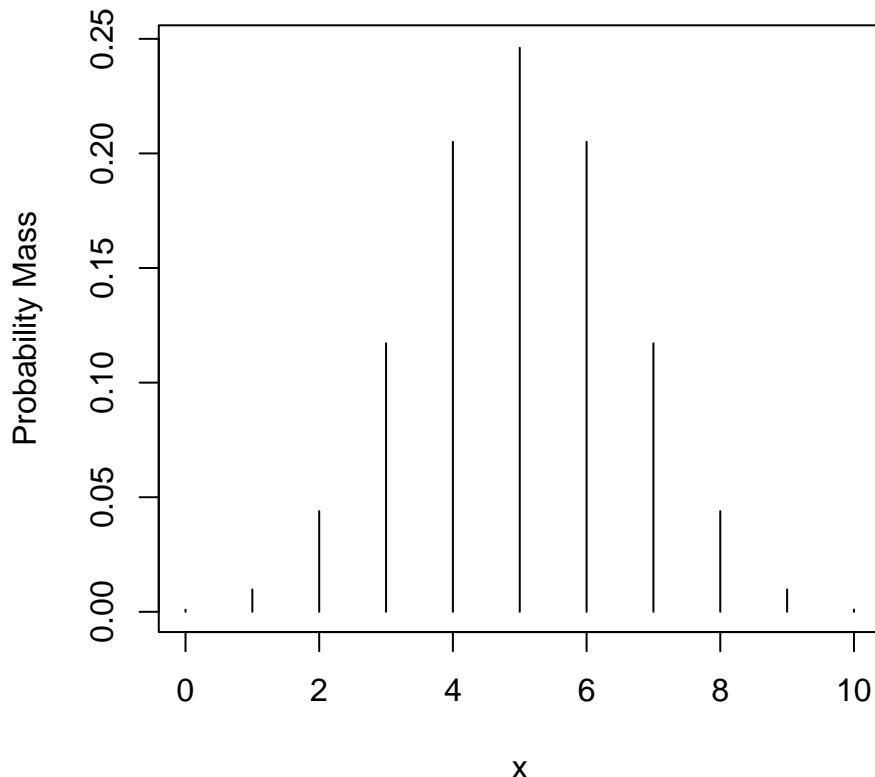
```
## [1] 0.2050781
```

Or to plot the distribution itself.

```
dbinom(0:10, 10, 0.5)

## [1] 0.0009765625 0.0097656250 0.0439453125 0.1171875000 0.2050781250
## [6] 0.2460937500 0.2050781250 0.1171875000 0.0439453125 0.0097656250
## [11] 0.0009765625

plot(0:10, dbinom(0:10, 10, 0.5), type = "h", xlab = "x", ylab = "Probability Mass")
```



4.3.1.2 dnorm()—continuous distribution

Consider a normal distribution with mean 0 and standard deviation 1.

The R function `dnorm()` gives the probability density at a given value of x .

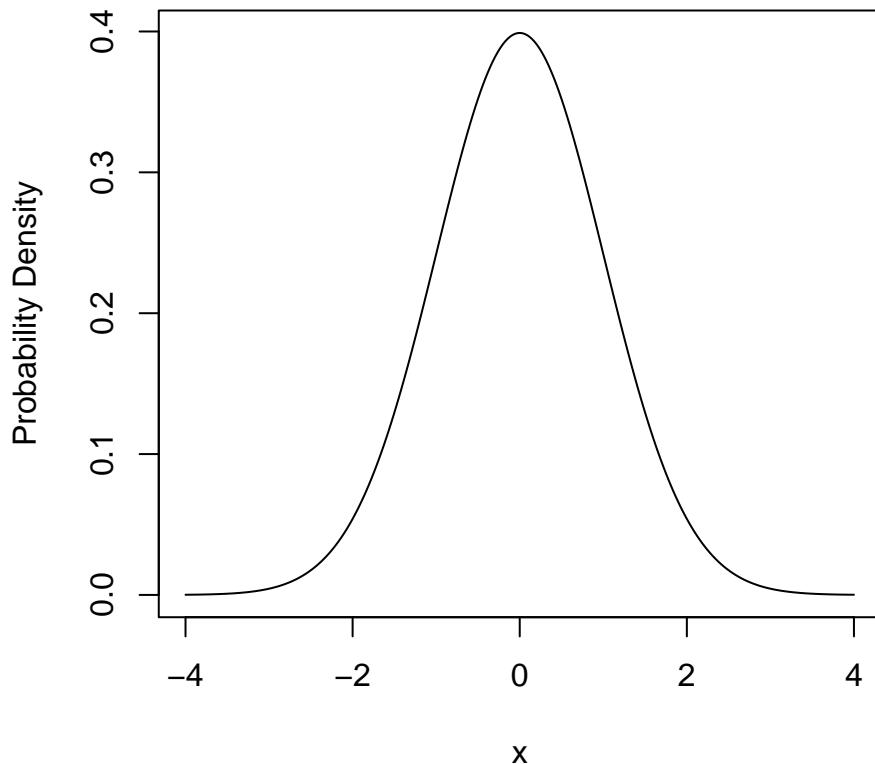
It takes the following arguments:

- `x`
- `mean`
- `sd`: standard deviation

The function `dnorm(x, mean, sd)` isn't useful in itself but is useful for plotting.

With a continuous distribution we have to choose the x coordinates close enough together to approximate a smooth curve.

```
x <- seq(-4, 4, 0.01)
plot(x, dnorm(x, 0, 1), type = "l", xlab = "x", ylab = "Probability Density")
```



4.3.2 pdist()

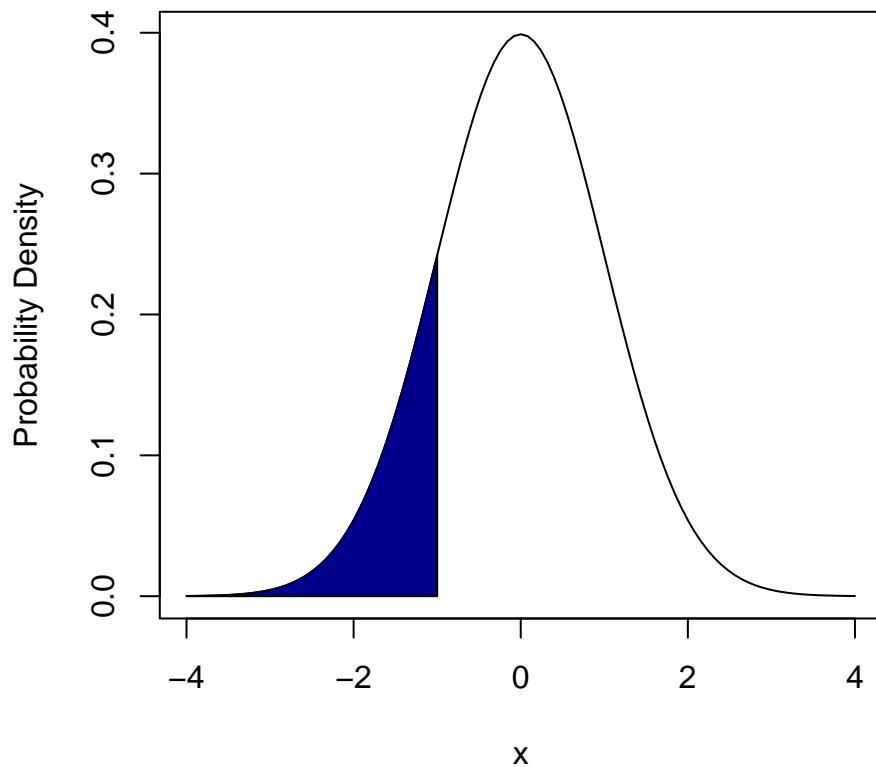
For a distribution dist `pdist(x, ...)` returns the probability $P(X \leq x)$ (which in a continuous function is an actual probability).

4.3.2.1 `pnorm()`—probability calculations

If we want to find the probability of having a value below -1 from a normal distribution we use `pnorm()`.

```
pnorm(-1, 0, 1)
```

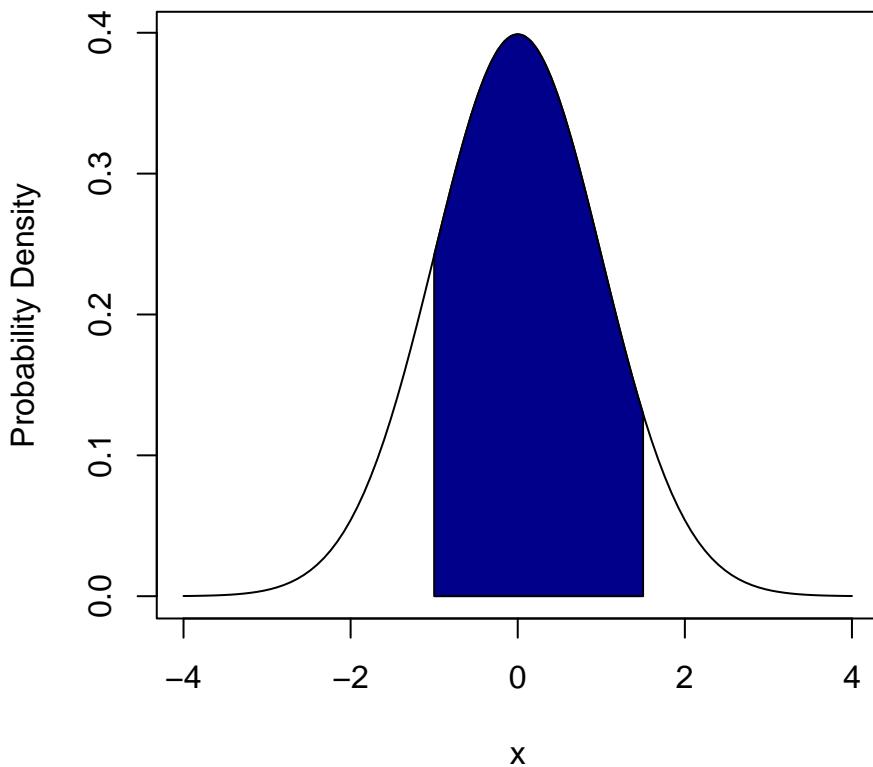
```
## [1] 0.1586553
```



If we want to find the probability of having a value between -1 and 1.5 from a normal distribution we use `pnorm()`.

```
pnorm(1.5, 0, 1) - pnorm(-1, 0, 1)
```

```
## [1] 0.7745375
```



4.3.3 rdist()

For a distribution dist `rdist()` returns n random numbers drawn from that distribution.

This function is of great use when trying to run stochastic models. The ability to draw random numbers from many different types of distribution is incredibly useful.

4.3.3.1 rnorm()—normal distribution

Consider a normal distribution with mean and standard deviation.

The R function `rnorm()` returns random numbers drawn from this distribution.

It takes the following arguments:

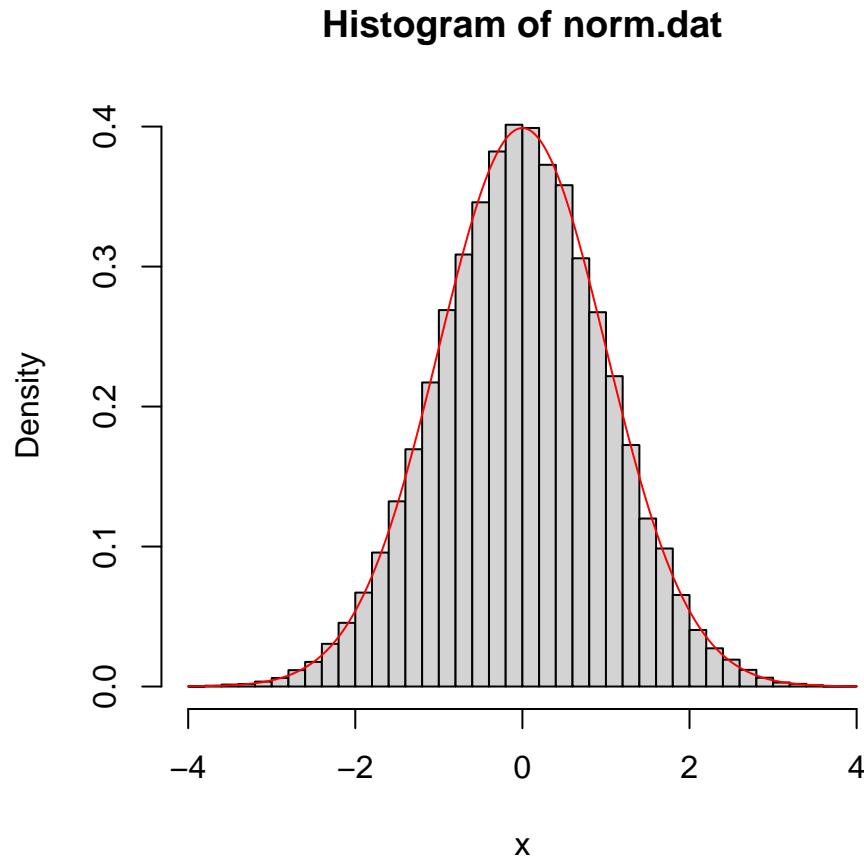
- `n`: number of random numbers to return
- `mean`
- `sd`: standard deviation

```
norm.dat <- rnorm(5, 0, 1)
print(norm.dat)
```

```
## [1] -1.1010736  0.2814474 -1.4025187  0.6546403  0.1511248
```

Consider a normal distribution with mean 0 and variable 1. We will draw thousands of random numbers from this distribution and plot a histogram of them. This should then look like the theoretical normal distribution curve we produce with `dnorm()`.

```
norm.dat <- rnorm(50000, 0, 1)
hist(norm.dat, prob = TRUE, breaks = 50, xlab = "x")
x <- seq(-4, 4, 0.01)
lines(x, dnorm(x, 0, 1), col = "red")
```



4.4 Example—using probabilities for simulation

Consider tossing a biased coin. There are two outcomes: Heads or Tails.

$$P(H) = \frac{1}{3} \text{ & } P(T) = \frac{2}{3}$$

How can we use probability distributions in R to simulate this?

4.4.1 Binomial distribution

Using the binomial distribution:

- 1 observation
- 1 trial
- Probability of success (H) is $\frac{1}{3}$

```
rbinom(
  1, # number of observations
  1, # number of binomial trials
  1/3 # binomial probability of success
)
```

```
## [1] 1
```

`rbinom()` returns the number of successes from our 1 observation of tossing a biased coin once.

- 0: $\frac{2}{3}$ of the time
- 1: $\frac{1}{3}$ of the time

We can check this by getting R to do many observations at once...

```
rbinom(15, 1, 1/3)
```

```
## [1] 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1
```

4.4.2 Uniform distribution

Using the uniform distribution:

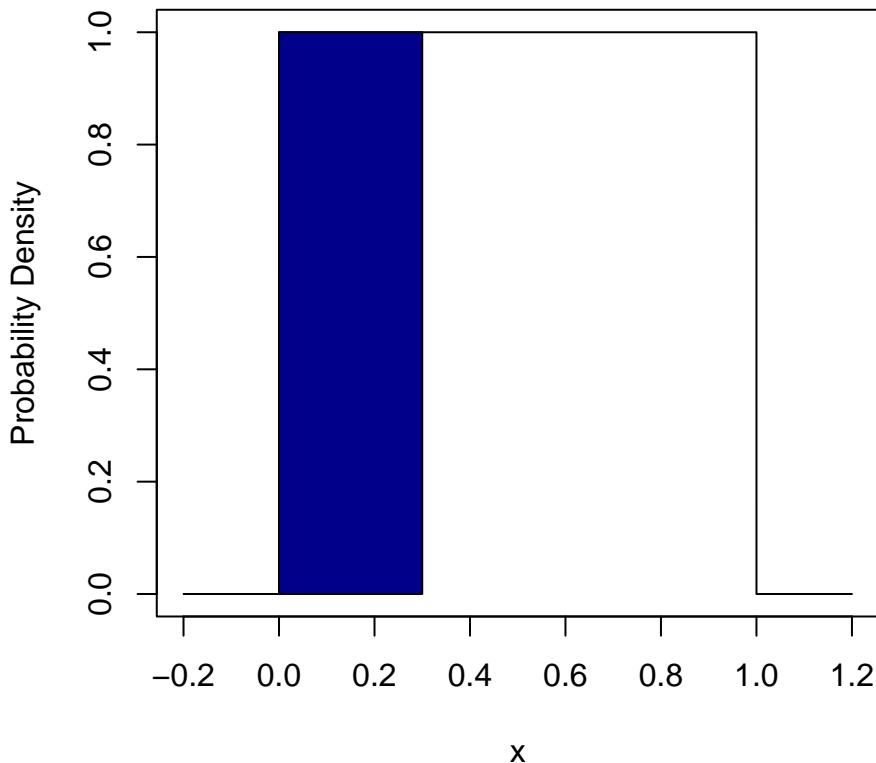
Generate single number uniformly between 0 and 1.

```
rand <- runif(
  1, # number of observation
  0, # minimum
  1 # maximum
)
if (rand < 1/3) {
  print("Heads")
} else {
  print("Tails")
}
```

```
## [1] "Tails"
```

```
print(rand) # confirm by printing rand
```

```
## [1] 0.3676546
```



The shaded area corresponds to the probability of drawing Heads.

4.5 Example—multiple choices

Consider tossing a very thick biased coin. There are three outcomes: Heads, Tails, or edge.

- $P(H) = \frac{3}{10}$
- $P(T) = \frac{6}{10}$
- $P(E) = \frac{1}{10}$

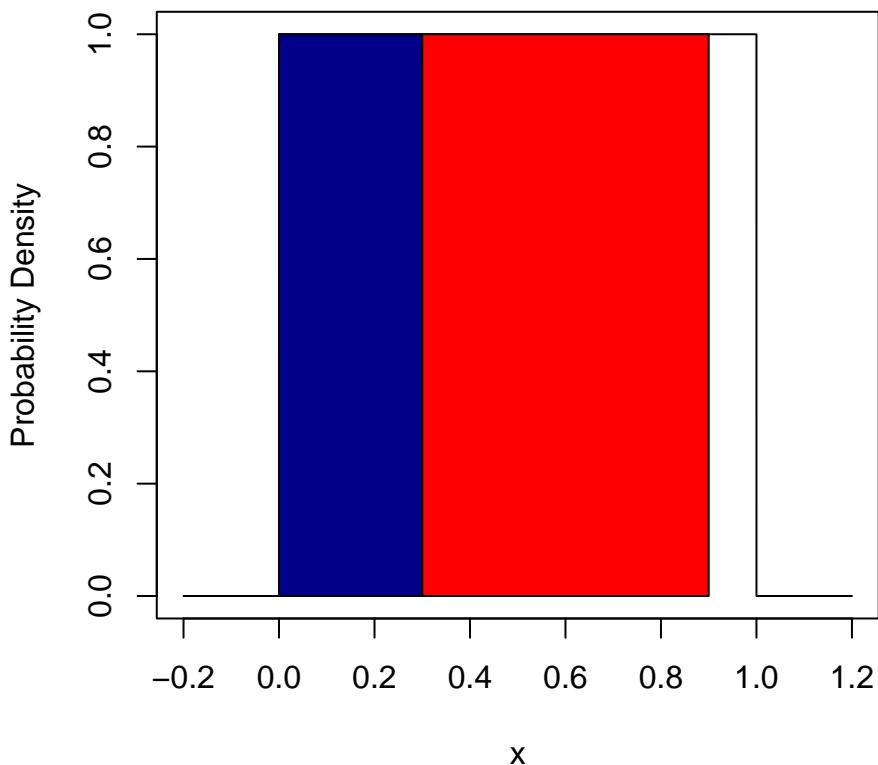
How can we use probability distributions in R to simulate this?

```
rand <- runif(1, 0, 1)
if (rand < 3/10) {
  print("Heads")
} else if (rand < 9/10) {
  print("Tails")
} else {
  print("Edge")
}

## [1] "Tails"

print(rand) # confirm by printing rand

## [1] 0.5052299
```



The blue shaded area corresponds to the probability of drawing Heads. The red shaded area corresponds to the probability of drawing Tails. The unshaded area corresponds to the probability of drawing Edge.

Task 33

Consider a regular six-sided die. How could we simulate a single throw of a die in R?

- What are the possible outcomes?
- Use `runif()`.
- Use (lots of) `if else` statements.

Show: Solution on P209

4.6 Example—using R to model a death process stochastically

Consider a population of animals. Every year each animal has a fixed probability of dying (depressing I know!), independently of each other and how old they are. There are no new animals born.

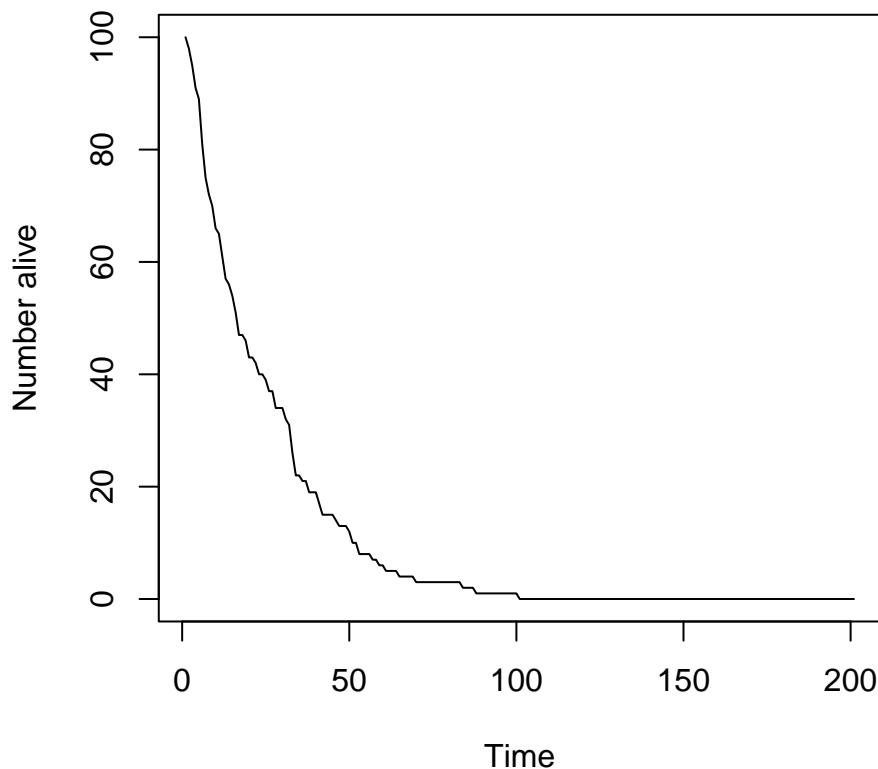
How could we model this?

- Need a program that keeps track of number of live animals.
- At each time step, we need to ‘roll a die’ to see if each live animal survives.
- We update the numbers of live animals and keep going until everything dies!

This is easy to do in R.

```
nAlive <- 100                                # current number of live animals
max.time <- 200                                 # keep going for up to 200 steps
nAliveStore <- numeric(max.time+1)               # vector to store how many were alive at each step
nAliveStore[1] <- nAlive                         # store how many were alive at the start
for (iTim in 1:max.time) {                        # loop over all time steps
  pr <- runif(nAlive, 0, 1)                      # probability of dying for all currently live animals
  died <- which(pr < 0.05)                       # find out which animals have died
  nDead <- length(died)                          # count up how many animals have died
  nAlive <- nAlive - nDead                        # update the number of currently living animals
  nAliveStore[iTim + 1] <- nAlive                # store the number of living animals
  if (nAlive <= 0) {break}                         # stop process when all animals are dead
}

plot(nAliveStore, type = "l", xlab = "Time", ylab = "Number alive")
```



Part III

Dynamic models

Chapter 5

Dynamic models—Practical

Katy Gaythorpe (k.gaythorpe@imperial.ac.uk)

This practical introduces the `deSolve` package in R.

5.1 A bit of theory

5.1.1 Rationale:

We have discussed the solutions of two differential equations using the fact that we were able to solve them analytically:

$$\frac{dN}{dt} = rN \rightarrow N(t) = N_0 e^{rt}$$
$$\frac{dN}{dt} = rN \left(1 - \frac{N}{K}\right) \rightarrow N(t) = \frac{K}{1 + \frac{K-N_0}{N_0} e^{-rt}}$$

However, most differential equations cannot be solved analytically. For example, with the predation model presented in Dynamic Models Lecture 2, we cannot obtain a mathematical formula giving the number of bacteria (prey) $N(t)$ and amoeba (predators) $P(t)$ as functions of the parameters and initial conditions. Instead we have to resort to numerical methods to obtain approximate values of $N(t)$ and $P(t)$ for any given set of parameter values. Since this is the case for the vast majority of differential equations of interest in science, such numerical methods have been developed for many years and the advent of computers have made them increasingly popular. Throughout this course we will be using a ready-made package in R called `deSolve` which can solve numerically virtually any system of ordinary differential equations (ODE).

Note

In earlier versions of R, the package was called `odesolve` but worked in the same way.

5.1.2 A few notes on Euler's method

Although we will be using `deSolve` as a black box, for your information here is a very basic introduction to how differential equation solvers work. The general principle was devised by Swiss mathematician [Leonhard](#)

[Euler \(1707–1783\)](#). Among many other things, Euler also came up with the fascinating formula $e^{-i\pi} + 1 = 0$, but that's another story.

Here I illustrate the method with a single differential equation, but it can easily be extended to any number. Consider the logistic growth model with an arbitrary choice of parameter values: $dN/dt = 0.5N(1 - N/100)$ and initial condition $N(0) = 1$.

The trick is then to ‘discretise’ this equation, i.e. replace the infinitesimal variations of the differential equation by small ‘discrete’ variations. Remember the formal definition of the derivative of a function:

$$\frac{dN}{dt} = \lim_{\delta t \rightarrow 0} \frac{N(t + \delta t) - N(t)}{\delta t}$$

We then approximate the variations during a small time step δt as follows:

$$\frac{N(t + \delta t) - N(t)}{\delta t} \approx 0.5N(t) \left(1 - \frac{N(t)}{100}\right)$$

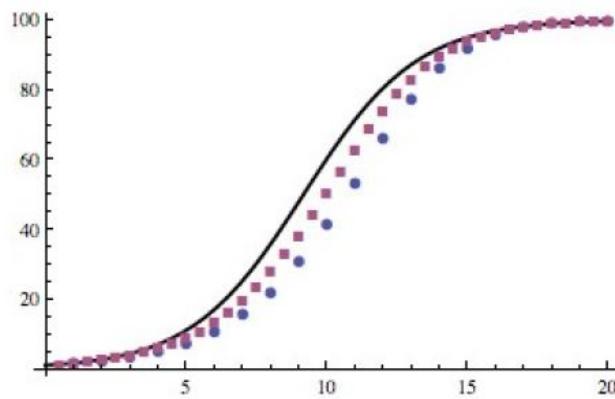
So, if we know the value of $N(t)$ at a given time t , we can estimate the value at the next time point:

$$N(t + \delta t) \approx N(t) + 0.5N(t) \left(1 - \frac{N(t)}{100}\right) \delta t$$

If you know the initial density, say $N(0) = 1$, and you choose $\delta t = 0.1$, then you can iteratively calculate approximate values of $N(0.1) \approx 1.0495$, $N(0.2) \approx 1.1014$, etc. which eventually enables you to reconstruct the numerical solution $N(t)$.

The only problem is that $N(0.2)$ is estimated based on an approximate value of $N(0.1)$, and so on. So, as you iterate the process, you accumulate errors and you run the risk of seeing your ‘numerical solution’ drift away from the true solution.

Of course, the smaller δt , the more accurate the discretisation. See the graph below, where the solid curve is the true solution of the above differential equation (which, in this case, can be solved analytically) and the dots are two ‘discretised solutions’, using $\delta t = 1$ (discs) and $\delta t = 0.5$ (squares):



So, that's basically what `deSolve` or any other software does to generate ‘numerical solutions’ of differential equations. They actually use more sophisticated algorithms (but based on the same ‘discretisation’ principle) in order to reduce the potential ‘drift’ from the (unknown) true solution. Different algorithms will perform better on different classes of models, but they are not 100% accurate. One of the most popular class of algorithms for the kind of ODEs used in population dynamics was designed by German mathematicians Runge & Kutta almost 100 years ago.

5.2 The return of the logistic model

5.2.1 Load the deSolve package

But enough theory for now. Let's shut the bonnet and sit behind the wheel! We'll start by re-visiting the logistic growth model to get you acquainted with `deSolve`.

Launch RStudio. The first thing to do is check whether the `deSolve` package is already installed, as it does not come with the R base. In RStudio, click on the “Packages” tab in the bottom-right window to see the list of R packages installed on your computer (those with a checked box are not only installed on the computer but currently loaded in your R session so that you can use them). If `deSolve` is not in the list, click on Install Packages, this opens a window asking you which package you want to install: type `deSolve` and click on Install (you may be asked first to choose a “Mirror”: find the mirror closest to where you are currently located, and select it). This will download the package from the internet, which means your computer needs to be connected. You only need to do that once on any computer.

Note: Instead of using RStudio’s interface, you can type the following instructions in the console: `library()` will open a window with the list of installed packages. If `deSolve` is not there, close the window, return to the R console and type: `install.packages('deSolve')`.

Once a package is installed on a computer, you still need to tell R to load the package into the memory every time you start a new session. In RStudio you can do that by checking the box of the package in the Packages window, but if you need to use the package in a script file, it is safer to type the command `library(deSolve)`.

We're now ready to start. Open a new script file for this practical code, write some comments (starting with a `#`) that will remind you what's in this file next time you open it, and type the first instruction:

```
library(deSolve)
```

REMEMBER TO SAVE YOUR FILE REGULARLY

5.2.2 Define the model

First, we must define a function `logistic_dyn()` that returns the value of the derivative dN/dt , in this case $rN(1 - N/K)$, given the values of r , K and N . This function will be used by the differential equation solver to generate the numerical solution $N(t)$, so we have to follow specific rules set by the authors of the `deSolve` package:

- The function that defined the derivative must take three arguments:
 - time,
 - a vector containing the values of the variables (in this case a single variable N), and
 - a vector of parameter values (in this case two parameters r and K).
- The function must return a list containing the value(s) of the derivative(s).

Note that the name of the function, `logistic_dyn`, is arbitrary; you may call it anything you like. Type the following code in your script file:

```
#This function calculates dN/dt for the logistic model

logistic_dyn <- function(t, N, par){
  #rename the parameters
```

```
r <- par[1]
K <- par[2]
#calculate the derivative
dN <- r*N*(1-N/K)
#last instruction: return a list
return(list(dN))
}
```

Note that before calculating the derivative, I defined a few local variables r , K and dN (which only ‘exist’ inside the function, so they do not appear in the Workspace) for convenience. A shorter version of the function, without local variables, would be:

```
logistic_dyn <- function(t, N, par) {
  list(par[1]*N*(1-N/par[2]))
}
```

but you’ll probably agree that it is less easy to read for a human being. The only case when you should favour the latter version is when speed (or memory) is an issue.

Next, create three variables:

- a vector of parameter values (2 values in the order specified in `logistic_dyn`: r first and K second):

```
logistic_par <- c(1, 100)
```

- a vector of time steps at which we want values of N (note that `deSolve` automatically computes many intermediate time steps as explained in the preamble):

```
logistic_t <- seq(0, 20, 0.1)
```

- the initial state of the system (in this case just the value of N_0):

```
logistic_init <- 1
```

We can then feed all that into a function called `lsoda()`, which is the only function from the `deSolve` package that we’re going to use, and store the result into a variable:

```
logistic_sol <- lsoda(logistic_init, logistic_t, logistic_dyn, logistic_par)
```

Save your file and run all your code so far (click on Source in the top right corner of the script window). In the Workspace window, you should now see the variables you’ve defined under Values, or your new function `logistic_dyn`, under Functions. To visualise the contents of the function, click on its name in the Workspace window, or type `fix(logistic_dyn)` in the Console: this opens a separate window that you must remember to close before you can re-gain access to the command line. You can also look at the contents of the matrix `logistic_sol` in the same way: the first column is your `logistic_t` vector (i.e. the time steps) and the second column contains the corresponding values of $N(t)$.

5.2.3 Version with named parameters

A feature of R that can make your life a bit easier when using `lsoda()` is the ability to name the elements of a vector. When we defined `logistic_par` earlier, we had to remember that the first element must be r

and the second one K . Not too bad with only two parameters, but it can be more of an issue with higher numbers of variables and parameters, as we'll see next in the next part.

Here is an alternative version of the function `logistic_dyn` which assumes that the third argument, `param`, is a vector whose elements are named r and K :

```
# Calculate the derivative dN/dt using a named vector as its third argument
named_logistic_dyn <- function(t,N,par) {
  # Tell R to use the names of the elements of par (r and K)
  with(as.list(par),{
    return(list(r*N*(1-N/K)))
  })
}
```

The syntax is a bit complicated because we have to tell R that r and K (which are undefined symbols) will actually be defined within `par`. That's the role of the `with(x,{...})` statement. A further complication is that `with(x,{...})` only works with a named list as its first argument, not a named vector, which is why we wrote `with(as.list(par),{...})`. As a result, every instruction that comes inside `{ }` in the second argument of `with(x,{...})` can make use of the names of the elements of `par`.

Once the function `named_logistic_dyn` has been defined, it can only be used with a named vector as a third argument. So, in our case, we must define a new vector:

```
named_logistic_par <- c(r=1, K=100)
```

Then you can use `lsoda` as follows:

```
named_logistic_sol <- lsoda(logistic_init,
                             logistic_t,
                             named_logistic_dyn,
                             named_logistic_par)
```

This will give you exactly the same result as `logistic_sol` above. So, with this method, you don't need to remember the order of the parameters in the function that calculated the derivative, but you need to remember their names.

5.2.4 Plot the dynamics

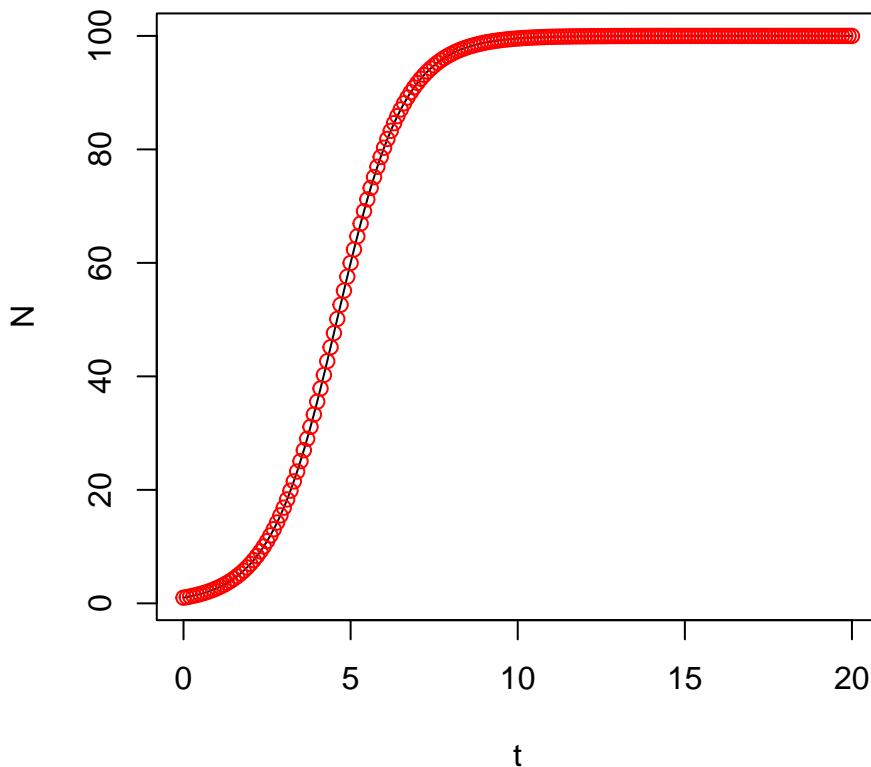
The next step is to plot the dynamics of $N(t)$ using the contents of `logistic_sol`:

```
plot(logistic_sol[,1], logistic_sol[,2], type='l',
     main='Logistic growth', xlab='t', ylab='N')

# Just to check, you can superimpose the graph of the
# analytical solution of the logistic growth model

points(logistic_t, 100 / (1 + (100 - 1) * exp(-logistic_t)), col = 'red')
```

Logistic growth



Task 34

Now plot a few more graphs with different parameter values (re-define `logistic_par`) or different initial values (`logistic_int`).

5.3 The revenge of the amoeba

5.3.1 Code the model

We'll follow the template described in the previous section, starting with the core function that calculates dN/dt and dP/dt . The only difference is that we now have 2 variables, so the function must return a list of two derivatives. There are 5 parameters, so you must be extra careful with the order in which you store them in the vector. To be safe, we'll enter them in alphabetical order:

```
# Definition of the predation model
predation_dyn <- function(t,var,par) {
  a <- par[1]
  b <- par[2]
  d <- par[3]
  K <- par[4]
  r <- par[5]
  N <- var[1]
  P <- var[2]
```

```

# Derivatives
dN <- r*N*(1-N/K)-a*N*P
dP <- -d*P+b*N*P
return(list(c(dN,dP)))
}

# Parameter values, initial variables values and the vector of time values:
predation_par <- c(a=0.02, b=0.01, d=0.3, K=100, r=1)
predation_init <- c(N=100, P=1)
predation_t <- seq(0,40,0.2)

# Numerical solution
predation_sol <- lsoda(predation_init,
                        predation_t,
                        predation_dyn,
                        predation_par)

```

The resulting matrix has 3 columns: `predation_sol[,1]` contains the time points, `predation_sol[,2]` the values of N and `predation_sol[,3]` the values of P .

Once again, we only defined the local variables a , b , d , etc. to make it easier to read the code and spot any mistakes. But we could write a more concise version using only `par[1]`, `par[2]`, etc. instead. Alternatively, you could use `with(as.list(c(par,var)), { ... })` as we saw [above](#).

5.3.2 Plot some graphs

Since we have two variables in this model, we can generate more diverse graphs. You should now be getting familiar with plotting instructions, so I'll let you work out how to produce the following graphs.

Task 35

Plot $N(t)$ and $P(t)$ against time, together on a graph, using `plot()`, `lines()` and `legend()`.

Show: Solution on P209

Task 36

Plot N and P in the phase plane. Use the notes from Lecture 2 to find the equations of the nullclines, and draw them using `abline()`. Finally add the equilibrium points.

Show: Solution on P211

5.3.3 Discussion and further investigations

Task 37

The model has three equilibrium points, determined by the values of the parameters. In the example above, the dynamics converged to the equilibrium point where the prey and predator coexist.

1. Do you think it would be possible to reach another equilibrium point from different initial conditions?
2. What do the dynamics look like if we start from $N(0) = 1$ and $P(0) = 100$?

Show: Solution on P212

Question 38

Is that what you expected from a biological point of view?

Show: Answer on P213

Question 39

What limitations of the model does this suggest?

Show: Answer on P213

Task 40

Observe the effects of changing values of b and d on the equilibrium points and the dynamics. What happens if $d > bK$?

Show: Solution on P213

Task 41

Change the values of b and d simultaneously so that the ratio b/d remains constant. What do you notice?

Show: Solution on P214

Part IV

Epidemic models

Chapter 6

Epidemic models—Practical 1

Katy Gaythorpe (k.gaythorpe@imperial.ac.uk)

6.1 The basic *SIR* model

Task 42

Write down the equations of a frequency-dependent *SIR* model with no births or deaths and transmission rate β and recovery rate γ .

Show: Solution on P215

Task 43

Show that for this model the population size remains constant, i.e. that $dN/dt = 0$.

Show: Solution on P215

6.2 Coding the model

We'll follow the template for `deSolve` that we used in the previous practical, starting with the core function that calculates dS/dt , dI/dt and dR/dt . Open R, create a new text files and type in:

```
# SIR Epidemic model - Practical 1
library(deSolve)

SIR_dyn <- function(t,var,par) {
  # Rename the variables and parameters
  S <- var[1]
  I <- var[2]
  R <- var[3]
  N <- S+I+R
  beta <- par[1]
```

```

gamma <- par[2]

# Derivatives
dS <- -beta*S*I/N
dI <- beta*S*I/N-gamma*I
dR <- gamma*I

# Return the 3 values
list(c(dS,dI,dR))
}

```

REMEMBER TO SAVE YOUR FILE REGULARLY

Note that in this model, the total number of individuals $S + I + R$ remains constant, so we would actually only need to define 2 variables (say S and I) and calculate 2 derivatives. I've included the 3 variables for clarity only. Then define parameter values, initial variables values and the vector of time values:

```

beta <- 1
gamma <- 0.25
SIR_par <- c(beta,gamma)
SIR_init <- c(99,1,0)
SIR_t <- seq(0,30,by=0.1)

# The numerical solution is given by
SIR_sol <- lsoda(SIR_init,
                  SIR_t,
                  SIR_dyn,
                  SIR_par)

```

If you like, you can relabel your variables so that they are easier to use:

```

TIME <- SIR_sol[,1]
S <- SIR_sol[,2]
I <- SIR_sol[,3]
R <- SIR_sol[,4]
N <- S + I + R

```

Note

You cannot label time as `time` in lower case letters because it is an inbuilt function in R.

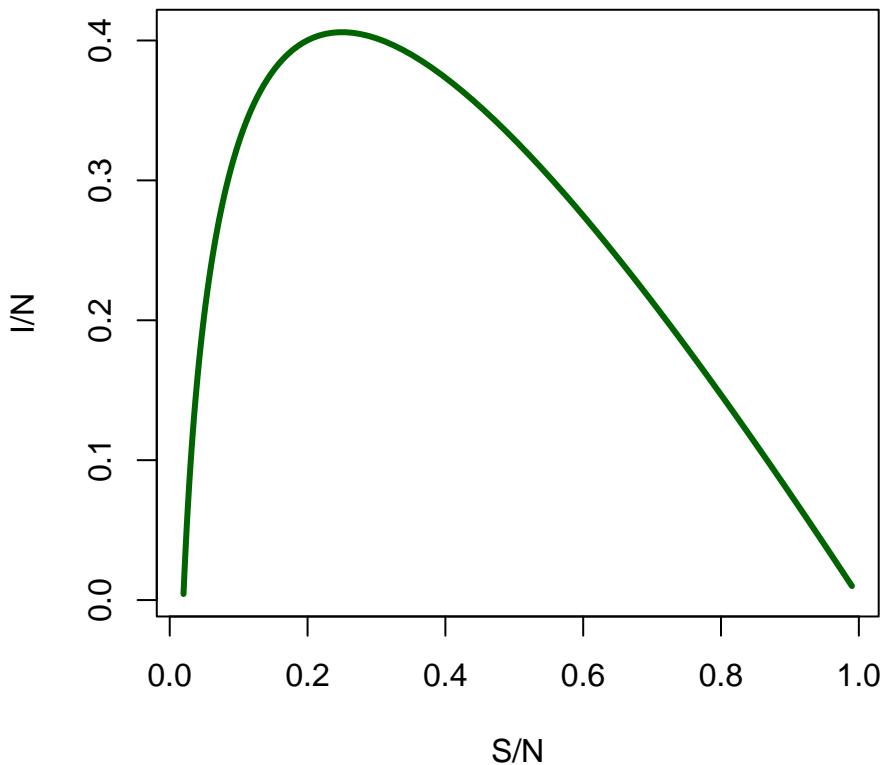
Task 44

Plot $S(t)$, $I(t)$ and $R(t)$ together on a graph, using the commands `plot()`, `lines()` and `legend()`.

Show: Solution on P216

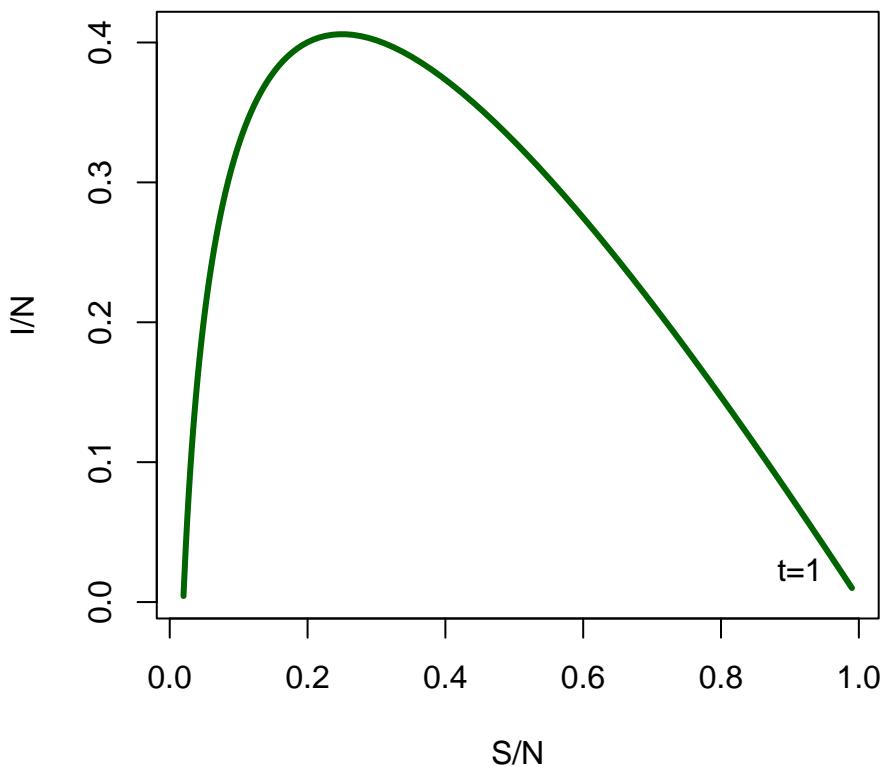
You don't have to always plot things against time, it can also be useful to plot variables against each other. Try plotting S against I :

```
plot(S/N, I/N, type = "l", lwd = 3, col = "darkgreen")
```



It can be difficult to read this graph at first. Find where $\text{TIME} = 1$ (at the start $S/N \approx 1$ and $I \approx 0$) and label it on the graph.

```
index <- which(TIME == 1)  
text(S[index]/N[index], I[index]/N[index], "t=1", pos=2)
```

**Task 45**

Also label $t = 2$, $t = 5$, $t = 10$ and $t = 15$. What is happening to time in this plot?

Show: Solution on P216

6.3 Some properties of the model

An important question is when does the epidemic reach its peak? A common misconception based on graphs similar to the one above is “ $I(t)$ reaches its maximum when $S(t) = R(t)$ ”.

By now you know that “ $I(t)$ reaches its maximum when $dI/dt = 0$ ”.

Task 46

- Given that $dI/dt = \beta SI/N - \gamma I$, when I is maximum we get $S/N = \gamma/\beta$.
- Check this relationship using the graph of S/N plotted against I/N . You might want to use `which.max()` which gives the position of the largest element in a vector.

Tip

You can use `abline` to mark on the plot where I/N reaches its maximum.

```
abline(v=S[which.max(I)]/N[which.max(I)], col ="black")
```

Next, let's look at the effects of β and γ on the dynamics of the system. In practice, β is usually not known before an epidemic, so it's estimated in the early stages of an epidemic using the average infectious period ($1/\gamma$) and R_0 . Instead of calculating and plotting a series of dynamics manually, we'll use a loop to plot 6 graphs in a single window.

Task 47

Disease	R_0	Infectious period, $1/\gamma$ (days)	β	Peak size	Peak time	Epidemic size
Flu (H1N1p)	1.2	1				
Flu	1.5	1				
SARS	2.0	28				
Smallpox	5.0	5				
Measles	17.0	7				
Pertussis	17.0	14				

Fill in the blanks and execute the following instructions:

```
# set up a 2 by 3 grid for the plots:
par(mfrow=c(2,3), xaxs='i', yaxs='i')
# define the parameters:
infperiod <- c(1,1,28,5,7,14)
Rzero <- c(1.2,1.5,2,5,17,17)
for (i in 1:6) {
  gamma = signif(1/infperiod[i],2)
  beta = signif(Rzero[i]*gamma,2)
  SIR_par <- c(beta,gamma)
  SIR_sol<-lsoda(SIR_init,
                  SIR_t,
                  SIR_dyn,
                  SIR_par)
  plot(....., ylim=c(0,1),type="l", main=paste0("beta = ",beta,"gamma = ",gamma))
  lines(.....) # I(t)
  lines(.....) # R(t)
}
par(mfrow = c(1, 1))
```

Show: Solution on P217

For each combination of parameter values, write down β and read off the approximate values of the height and time of the epidemic peak and the final epidemic size (the total number of individuals that were infected). (**Hint:** you may have to extend the time period!)

You can also use the functions `max()` and `which.max()`.

Remember from the lecture that $R_0 = \beta/\gamma$, therefore $\beta = \gamma R_0$.

Task 48

Which of the last 3 quantities appear linked to R_0 ?

Show: Solution on P218

Note:

It can be shown mathematically that the epidemic size and the height of the epidemic peak are functions of R_0 only, although this requires some clever manipulation of the ODEs (see [Appendix](#) at the end of this practical if you're interested).

The time to the epidemic peak cannot be directly expressed from the equations, but it is linked to the initial rate of spread of the epidemic. The variations in the number of infected individuals are given by:

$$dI/dt = I(\beta S/N - \gamma)$$

At the very beginning of the outbreak, $I(t) \ll S(t) \approx N$. So we can approximate the ODE above as $dI/dt = (\beta - \gamma)I$, which can be solved as:

$$I(t) = I_0 \exp([\beta - \gamma]t)$$

So the number of infected individuals initially follows an exponential growth with rate

$$\beta N - \gamma = \gamma(R_0 - 1)$$

Task 49

Plot $\log(I(t))$ for t in $[0, 4]$ for the same parameter combinations above and check that the initial gradient is equal to $\beta - \gamma$.

Tip

You can use `abline` again, this time to impose a straight line ($y = bx + c$).

Show: Solution on P219

Task 50

What dynamics do you expect with $R_0 = 1$? Plot the graph for $\beta = 1$, $\gamma = 1$. Why does $I(t)$ decrease?

Show: Solution on P220

Task 51

Explore the behaviour of the system around $R_0 \approx 1$ using the loop you wrote before. Try changing $0.9 < R_0 < 1.1$ and altering the initial number of individuals infected.

Show: Solution on P220

6.4 Calculating the infectious period

Question 52

If the recovery rate of a disease is $\gamma = 0.2 \text{ days}^{-1}$, what is the average infectious period?

Show: Answer on P221

- For a given constant recovery rate of $\gamma = 0.2 \text{ days}^{-1}$ we can plot the proportion of individuals still infected over time.

```
# set up a vector of times:
x=seq(0,30,0.1)

# define the recovery rate, gamma:
gamma=0.2

# define the exponential function:
fx=exp(-gamma*x)

# plot:
plot(x,fx,type="l")
```

- We can also calculate the average infectious period using the function defined above.

```
print(sum(fx*x)/sum(fx))
```

Task 53

How do your two answers compare?

Show: Solution on P222

6.5 Optional section—extending SIR to other compartments

As we saw in the lecture, the *SIR* framework is only a crude representation of the real natural history of a disease. Think of another formulation and see how the results compare to a standard *SIR* model. Some suggestions:

- Include loss of immunity, so that Recovered individuals become Susceptible again after a certain period of time.
- Include a latent state post-infection but before an individual becomes infectious (this type of model is often called an *SEIR* model, where *E* stands for the Exposed class).
- Include treatment in the model, allowing Infected individuals to recover more quickly.
- Sketch a diagram of your model, showing the compartments and the flows between them.
- Write down the equations that govern your model
- Write a function, such as `SEIR_dyn()` that calculates the derivatives of your model for use with `lsoda()`.

- Plot the epidemic curve, $I(t)$. How does this compare to an equivalent *SIR* model?

Show: Solution on P222

6.6 Appendix—Epidemic peak and final epidemic size

Although it is not possible to solve the differential equations of the *SIR* system analytically to obtain expressions for $S(t)$, $I(t)$ and $R(t)$, we can obtain an implicit solution in the (S, I) phase plane. In other words, we can obtain a mathematical relationship between the values of S and I at any time point. The trick is to re-write the system

$$\begin{aligned}\frac{dS}{dt} &= -\beta SI/N \\ \frac{dI}{dt} &= \beta SI/N - \gamma I\end{aligned}$$

as a single differential equation for I as a function of S . Under certain mathematical conditions that we won't discuss here, the variations of I with respect to S can conveniently be written as:

$$\frac{dI}{dS} = \frac{dI/dt}{dS/dt} = \frac{\beta SI/N - \gamma I}{-\beta SI/N} = \frac{\gamma N}{\beta S} - 1$$

Using the integration rules from the maths lectures, we can actually solve this ODE:

So, given the initial conditions, we can deduce the value of $I(t)$ from the value of $S(t)$ at any time point. How is that helpful? First consider the question of the height of the epidemic peak. We know that $I(t)$ reaches its maximum when $S(t) = \gamma/\beta$. Hence the height of the epidemic peak:

$$I_{\max} = I_0 + \frac{\gamma}{\beta} \log \left(\frac{\gamma}{\beta S_0} \right) + S_0 - \frac{\gamma}{\beta}$$

Assuming that $I(0) \ll S(0) \approx N$, then we can write $R_0 \approx \beta S(0)/\gamma$, hence:

$$I_{\max}/N \approx 1 - (1 + \log R_0)/R_0$$

You can check it on the graphs from section 2 above. Next, the final epidemic size is given by the final value of $S(0) - S(t)$, i.e. taking $t \rightarrow \infty$. Under the same assumptions as above, and letting $x = S(0) - S(\infty)$ be the proportion of the N population that gets infected over the whole epidemic, we get the following equation: $x + \log(1 - x)$, which can only be solved numerically.

R does not have a built-in numerical solver for nonlinear equation, but it has a function called `optimize()` that searches for the minimum of a numerical function. The trick is to notice that solving an equation of the form $f(x) = 0$ is equivalent to finding the minimum of $f^2(x)$, the square of $f(x)$. If you're not convinced, sketch a graph of an arbitrary function that crosses the x-axis, then sketch the square of that function.

We can therefore estimate the epidemic size for any value of R_0 using the following code:

```
my_fun <- function(x,R0) {
  (x+log(1-x)/R0)^2
}
optimize(my_fun, c(0,1), R0=2)
```

where the three arguments of `optimize` are:

- the function to minimize with respect to its first argument (in this case x),
- the interval over which to search for a solution
- the numerical values of any other argument of `my_fun` (in this case R_0).

Chapter 7

Epidemic models—Practical 2

Andrew Conlan (ajkc2@cam.ac.uk), Sophie Ip, Ellen Brooks Pollock (ellen.brooks-pollock@bristol.ac.uk)

7.1 Outline

In this practical we will:

- Code and run a model with vaccination.
- Investigate the critical vaccination threshold.
- Code and run a two-population model and investigate targeted vaccination.

7.2 Vaccination at birth

Consider a vaccine that confers complete protection against infection for life that is administered at birth. You will need to adapt your SIR model from practical 1 to include births, deaths and a permanent vaccination campaign where a proportion $p \leq 1$ of all newborns are immunized.

Task 54

Sketch out the model. **Hint:** You don't necessarily have to include a vaccinated compartment as you could include vaccinated people in the 'recovered' compartment.

Show: Solution on P223

Task 55

Write down the modified equations.

Show: Solution on P224

Task 56

Adapt your SIR model with births and deaths to include vaccination and run the model with parameters $\beta = 2$, $\gamma = 0.5$, $\mu = 1/70$, $p = 0$ and initial conditions $S(t = 0) = 999$, $I(t = 0) = 1$ and $R(t = 0) = 0$.

Show: Solution on P224

Task 57

What's the critical vaccination threshold for these parameters, p_{vac} ?

Show: Solution on P225

Task 58

Investigate increasing the proportion of vaccinated newborns from 0 to 1. What happens to the number of infected individuals when p crosses the critical vaccination threshold?

Show: Solution on P226

Question 59

Why do you still get an initial epidemic?

Show: Answer on P227

Task 60

Plot a horizontal line to indicate $1/R_0$. Notice what happens to the number of susceptible when $p = 0.75$. What about when $p = 0.7$?

Show: Solution on P228

Task 61

Modify your model to include a one-off vaccination campaign at the start so that there is no initial epidemic.

Show: Solution on P229

Question 62

What does this tell you about vaccination at birth programmes used in isolation?

Show: Answer on P233

7.3 Targeted Vaccination

Now we will consider vaccination in a population with two age groups. We assume that the epidemic dynamics are much faster than demographic changes, so the impact of births and deaths is negligible. Firstly, code up a two-population model. You can base it on the code from the earlier SIR practical.

```
# This proposed version uses some helpful features of the R language to simplify the code:
# "with" allows us to bring named members of a vector into the local namespace
# here we use to avoid having to specifically rename the elements of par
# We use the matrix/vector functionality to avoid writing down individual equations for
# each group. We don't need to do this it would just be longer to write out each equation in
# turn. You should try this as an excercise and check your new function gives the same dynamics!

TwoPopSIR <- function(t,states,par) {

  with(as.list(c(states, par)), {

    pops<-matrix(states[1:(npops*nstates)], 
    nrow=nstates,ncol=npops,byrow=T)

    S=pops[1,]
    I=pops[2,]
    R=pops[3,]

    N=colSums(pops)

    newinfections = colSums(beta*(I/N) %*% t(S))
    dS <- -newinfections
    dI <- newinfections - gamma*I
    dR <- gamma*I
    return(list(c(dS,dI,dR)))
  })
}
}
```

Run the model with initial conditions `init <- c(49,50,1,0,0,0)` and parameters $\gamma = \frac{1}{4}$ and $\beta = \begin{pmatrix} 1 & 0.1 \\ 0.1 & 0.5 \end{pmatrix}$.

```
# states are recorded in order (S1,S2,I1,I2,R1,R2)
# so we have 1 infected in patch 1 for initial condition, 50 individuals in both patches

TwoPop.init<-c(49,50,1,0,0,0)

# Note we have been lazy here and have written the function
# to use the "beta" matrix from the global workspace. This is bad programming form
# but again avoids messing about converting the matrix to and from vectors...

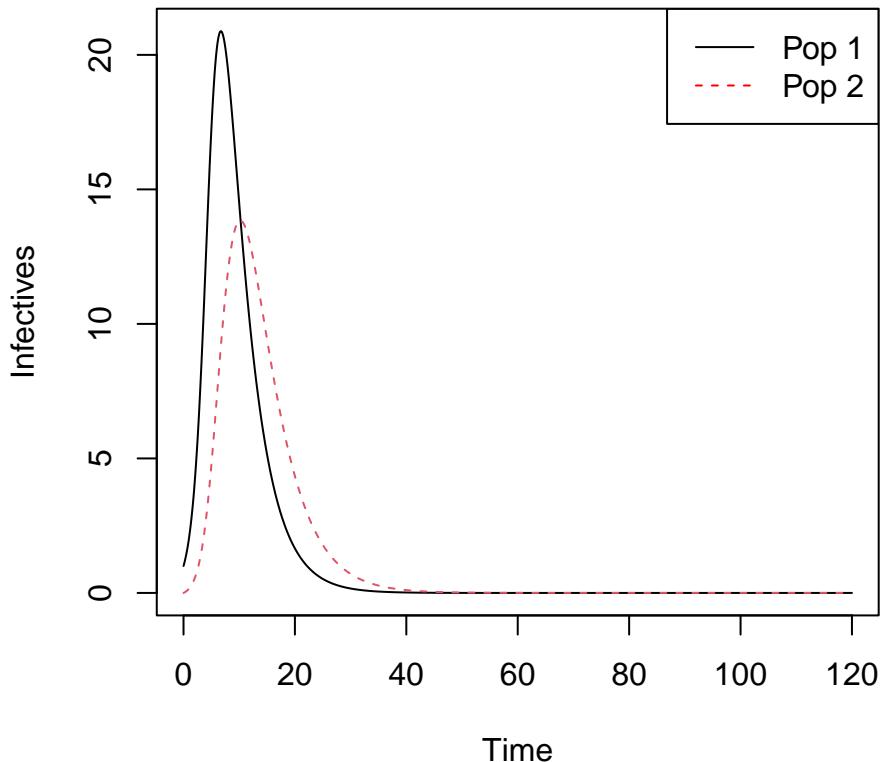
beta = matrix(c(1,0.1,0.1,0.5),2,2)

# As transmission parameters coded in beta above, one other parameter (named for use with with above).
TwoPop.par<-c(gamma=1/4.0,nstates=3,npops=2)

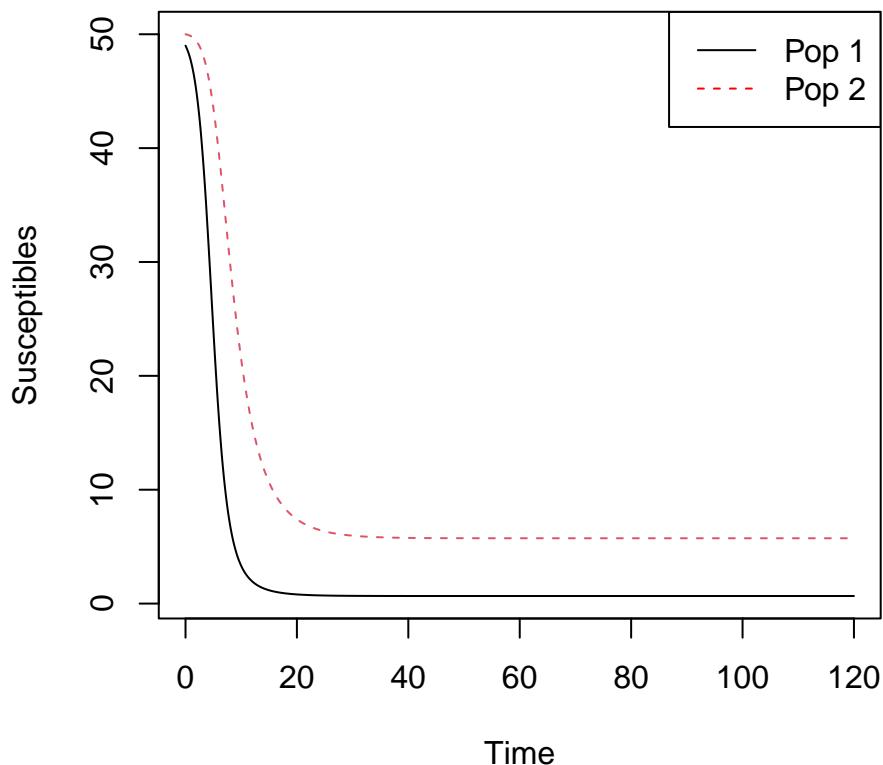
TwoPop.t<-seq(0,120,0.1)
```

```
TwoPop.sol <- lsoda(TwoPop.init,TwoPop.t,TwoPopSIR,TwoPop.par)

matplot(TwoPop.sol[,1],TwoPop.sol[,c(4,5)],type='l',xlab='Time',ylab='Infectives')
legend('topright',c('Pop 1','Pop 2'),col=c('black','red'),lty=c(1,2))
```



```
matplot(TwoPop.sol[,1],TwoPop.sol[,c(2,3)],type='l',xlab='Time',ylab='Susceptibles')
legend('topright',c('Pop 1','Pop 2'),col=c('black','red'),lty=c(1,2))
```



Note that the epidemic peak occurs later, and is lower, for group 2.

Task 63

Write down the next generation matrix for these parameters and calculate R_0 .

Show: Solution on P233

Task 64

What is the critical vaccination threshold for this disease?

Show: Solution on P233

Let's assume that a proportion p_1 of group 1 and a proportion p_2 of group 2 are vaccinated at the start of the model. The initial conditions are:

```
p1=0.8
p2=0.8
init <- c(49*(1-p1),50*(1-p2),1,0,49*p1,50*p2)
```

Task 65

By changing the initial conditions, explore the dynamics with random vaccination, where $p_1 = p_2$.

Show: Solution on P233

Task 66

Using the formula from the lecture, plot the total proportion of the population that needs to be vaccinated to prevent an epidemic $\frac{(p_1+p_2)}{2}$ for different values of p_1 .

Show: Solution on P235

Chapter 8

Epidemic models—Practical 3

Matt Castle (mdc31@cam.ac.uk) and Andrew Conlan (ajkc2@cam.ac.uk)

8.1 Stochastic models

Aim: To explore the effects of stochasticity on simple *SI* and *SIR* models and contrast their behaviour with previous deterministic results.

Goals:

By the end of this practical participants should be able to achieve the following:

- Code up a simple stochastic *SI* model in R using the `SimInf` package.
- Code up a simple stochastic *SIR* model in R using the `SimInf` package.
- Run both single and multiple realisations and visualise the results.
- Explore extensions to both *SI* and *SIR* models:
 - Expand the *SI* model to include additional transitions.
 - Understand the impact of R_0 values on stochastic extinction.

Background: This practical follows on from the Stochastic Epidemic Models Lecture and echoes the content presented there.

8.2 Coding a simple *SI* model

We will first implement a slight modification to the code that has been presented to you in the previous Stochastic Epidemic Lecture. The key difference between the code presented in the lecture and the code shown here is that we are putting the code into a function (called `StocSI.dyn`) rather than just running a series of commands.

- Open Rstudio.
- Open a new blank script file.
- Save it as `Stoch_SI.R`.

In the script window type the following commands:

```
library(SimInf)

create.StocSI <- function(N, I0, beta, f_time) {

  initial_state <- data.frame(S = N-I0 , I = I0)
  compartments <- c("S", "I")
  transitions <- c("S -> beta*S*I/(S+I) -> I")
  tspan <- seq(from = 1, to = f_time, by = 1)

  model <- mparse(transitions = transitions, compartments = compartments,
    gdata = c(beta = beta), u0 = initial_state, tspan = tspan)

  return(model)
}
```

Make sure you save your file and that your working directory is the same as the folder you have saved this into (Choose from menu Session > Set Working Directory > To Source File Location).

This code creates a function that creates a single stochastic *SI* model. We'll dissect/explain the code in the next section.

Open a new blank script file.

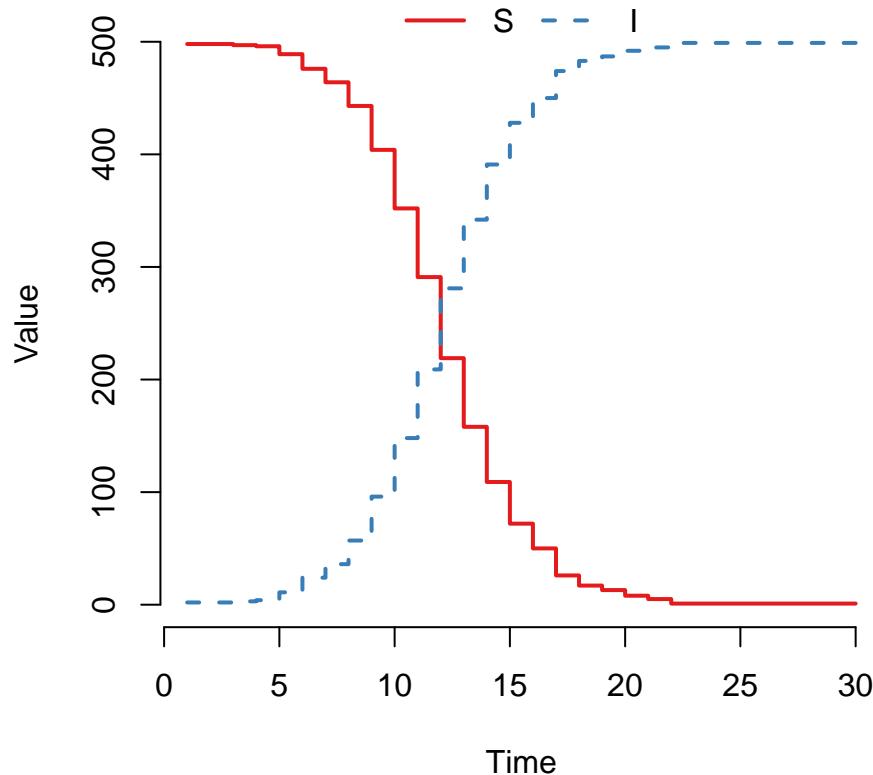
Save it as `Epi_P3.R`

We'll use this script to record all of the steps in this practical (unless otherwise indicated).

In the script window type the following commands and then run (select and press run button in RStudio):

```
source('Stoch_SI.R')
SImodel <- create.StocSI(500, 1, 0.5, 30)
out <- run(model = SImodel)
plot(out)
```

You should see a graph that looks similar to this:



Here we can see the two trajectories for an *SI* epidemic. The susceptible curve (red) starts at 499 and stochastically decreases down whilst the infected curve (blue) starts at 1 and mirrors the S curve increasing stochastically.

8.3 Dissecting the stochastic *SI* code

The function we have just created is a wrapper that takes the basic parameters that define the *SI* model for a particular population and creates a R object that the **SimInf** package can use to simulate the model. Our function `create.StocSI` takes four inputs:

- Total population size: `N`
- Initial number of infected: `I0`
- Infection rate: `beta`
- Time to simulate for: `f_time`

Inside the function we combine this information to create four R objects that when passed to the **SimInf** function `mparse`. The output from the `mparse` function is returned as the output to our `create.StocSI` function.

Here we will break down what we have just written and explain what is required for the `mparse` function.

Note

The **SimInf** package includes its own function that defines *SIR*, *SEIR* and *SIS* models (of which the *SI* model is a special case). The `mparse` function can define any generic compartmental model so it is worth learning about.

The `mparse` function requires that you specify the following information:

- the list of all compartments of the system
- the initial states (or compartments) of the system
- the events (or transitions) of the system
- the values of any parameters
- the time points where the value of each state is to be recorded for each simulation

An *SI* model is described by the following system of *differential* equations:

$$\frac{dS}{dt} = -\beta \frac{SI}{N} \quad (8.1)$$

$$\frac{dI}{dt} = \beta \frac{SI}{N} \quad (8.2)$$

And from these equations we can derive a transition diagram from the system:

$$S \rightarrow \beta \frac{SI}{N} \rightarrow I$$

- From this diagram we can see that there are two states for this system: susceptible and infected (*S* & *I*). We define all of the compartments of the model using a vector:

```
compartments <- c("S", "I")
```

- In order to begin the simulation, we need to specify the initial values of each compartment. We specify these using a data frame with a single row and column for each compartment specified above:

```
initial_state <- data.frame(S = N-I0, I = I0)
```

where *N* (the total population number) and *I0* (the initial number of infected) have been specified by the user.

- There is one event (or transition): infection.

```
transitions <- c("S -> beta*S*I/(S+I) -> I")
```

where the effect is to move one susceptible into the infected compartment at a rate $\beta \frac{SI}{N}$ where $N = S + I$ and *beta* must be specified by the user. The syntax we use here should be self-explanatory (hopefully) but the idea is that we have the following syntax “A → rate → B” where A is the name of the starting compartment, B is the name of the ending compartment and rate explains how to calculate the rate.

- We must tell algorithm how long we want it to run for before stopping and at what points we wish to record the value of the compartments. We use the user specified variable *f_time* to specify a sequence of daily output values up to a maximum value of *f_time*.

```
tspan <- seq(from = 1, to = f_time, by = 1)
```

Note

`SimInf` was designed to work with real epidemiological data and as such the minimum output timestep size is 1 (corresponding to 1 day).

- The `mparse` function takes the arguments we've just created, processes ('parses') them into a low-level computer language and compiles the resulting program into machine code. This machine code is considerably (orders of magnitude) faster than the equivalent R code. We assign the output of this function to the object `SImodel` (an object in R is a special data type that combines functions with data).
- Objects can be used as arguments to functions in the same way as normal data types. In this case we can use the `run()` function to run a stochastic simulation of the `SImodel`:

```
out <- run(model = SImodel, threads = 1)
```

The `threads` argument is optional here, but this argument would allow us to parallelise our code (if our setup allows it).

- The `run` function implements (and runs) the stochastic simulation algorithm for any stochastic model defined by the `mparse` function and returns a new object which contains the output of the simulation (which we've called `out`).
- The `SimInf` package provides a special plot function for this output

```
plot(out)
```

This explains the code you've produced so far, but there are limitations. We would like to plot multiple stochastic realisations onto a single plot and investigate their aggregated properties. To do this we will need to modify our code somewhat.

8.4 Performing multiple simulations

Let's update our `create.StocSI` function in the (`Stoch_SI.R` file) to create a model with multiple copies of the same population. This is the simplest possible meta-population model where there is no interaction between patches, and a neat hack to run multiple replicate simulations at the same time.

Edit `Stoch_SI.R` to update the definition of the `create.StocSI` as follows:

```
create.StocSI <- function(N, I0, beta, f_time, reps = 1) {
  initial_state <- data.frame(S = rep(N-I0, reps), I = rep(I0, reps))
  compartments <- c("S", "I")
  transitions <- c("S -> beta*S*I/(S+I) -> I")
  tspan <- seq(from = 1, to = f_time, by = 1)

  model <- mparse(transitions = transitions, compartments = compartments,
    gdata = c(beta = beta), u0 = initial_state, tspan = tspan)
  return(model)
}
```

The only two things we've changed here are:

1. We've created a new argument call `reps` which we're using to specify the number of simulations that we want the code to run.

2. We've changed how we specify the `initial_state` datafram. This is now a datafram that has `reps` number of rows and 2 columns. The first column contains the starting number of susceptible individuals in each of our simulations, and the second column contains the number of initial infected individuals.

Again, remember to **save** your file.

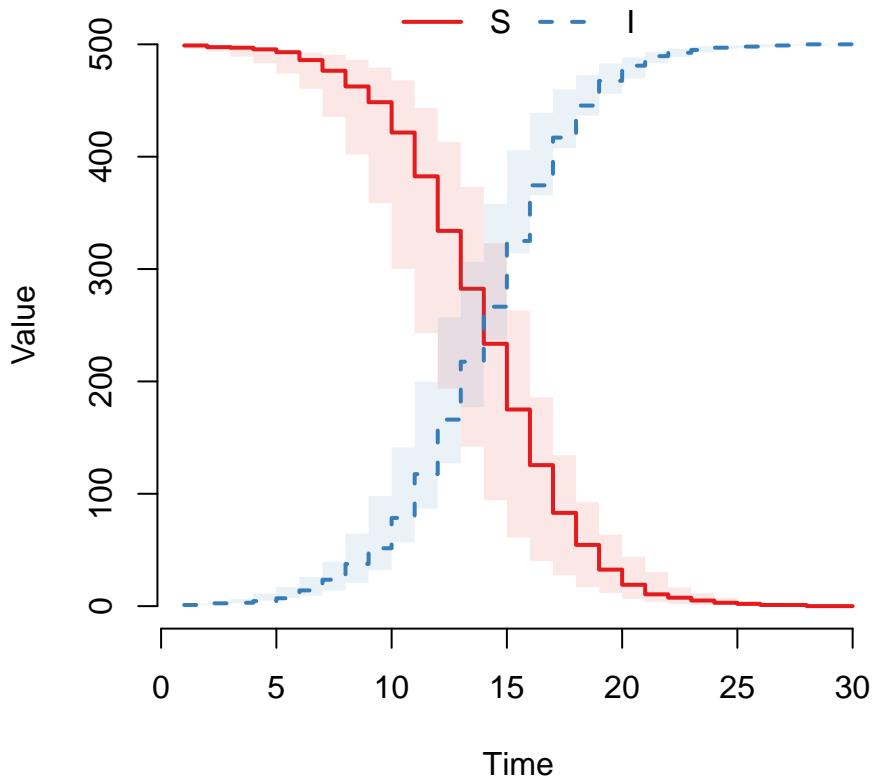
Note

We have added a new argument to `create.StocSI` which has a default value `reps = 1`. By adding the argument in this way we ensure that any code we wrote earlier assuming only one replicate would still work with no modifications.

Type in the following code into `Epi_P3.R` and run:

```
#Multiple replicates
source('Stoch_SI.R')
SImodel <- create.StocSI(500, 1, 0.5, 30, 20)
out <- run(model = SImodel)
plot(out)
```

This should produce a plot that looks similar to the following:



By default the plot command will always:

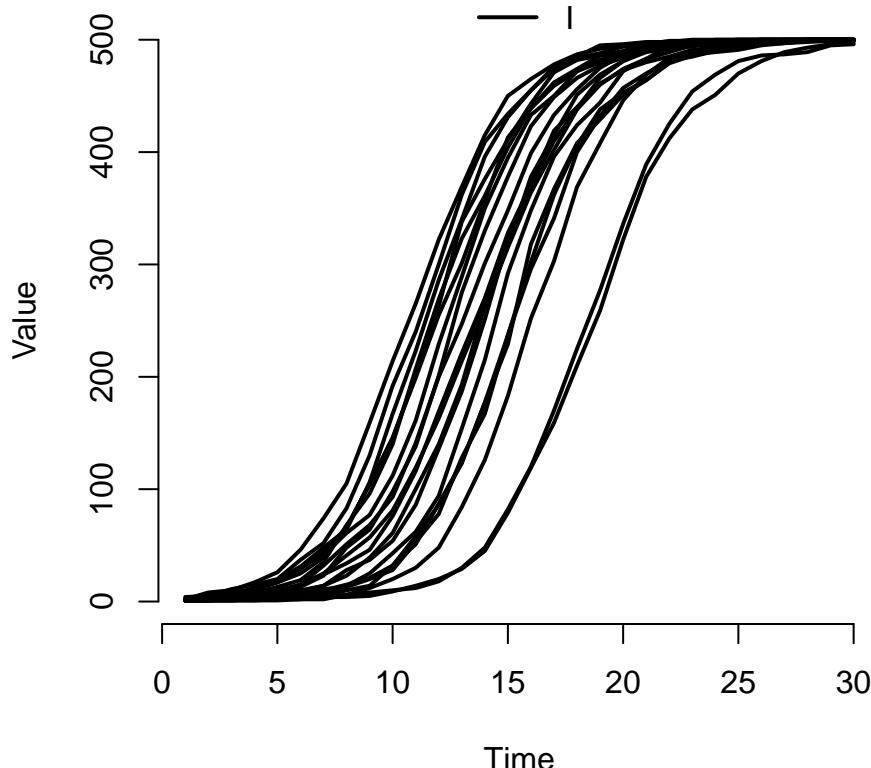
1. plot the curves for all states on the same graph (so we can see both S and I curves here)
2. use a step function to plot the curves (which is why it's a bit, um, step-y)

3. show the average and interquartile range as a transparent envelope when multiple simulations have been run.

If we want to actually see individual realisations, we need to use some of the other arguments to the plot function. **Try typing this in:**

```
plot(out, 'I', range = FALSE, type = 'l')
```

This should produce a plot that look similar to the following:



- the second argument is a character vector specifying which compartments we want to see. Here we've asked to only see the curves for the infected *I* compartment
- the named argument `range` is used to specify the shaded quantile on the plot. By specifying `FALSE`, we've turned it off and forced R to plot the individual trajectories instead. If we'd specified a numeric value between 0 and 1 then we would see the equivalent shaded range plotted instead (i.e. `range = 0.5` corresponds to plotting the interquartile range and `range = 0.95` corresponds to plotting the wider 95% quantiles)
- we can use the standard `type` argument from the base R plot function to specify the type of curve we want. The default is `type = 's'` for steps, but here we're asking for the smoother `type = 'l'` lines option.

8.4.1 Adding the deterministic solution curve

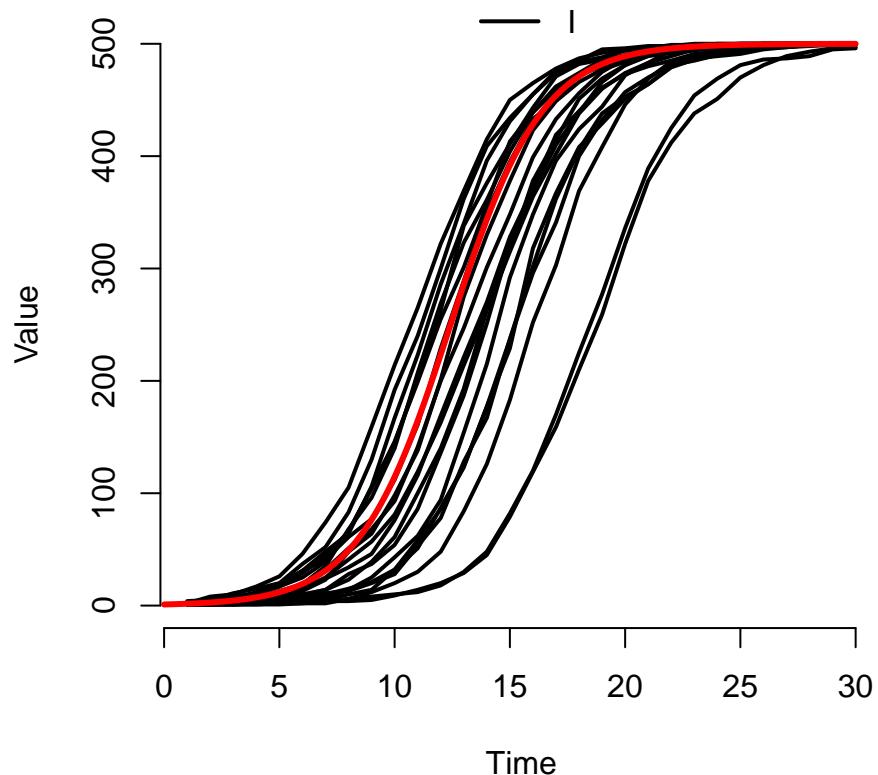
Download an R script file called `Det_SI.R`. It contains the code for a deterministic *SI* model based upon the code you encountered in the first epidemic models practical. It is specifically designed to use the same parameters as have been used in the stochastic *SI* code written above and will add a deterministic trajectory to your plot.

Type in the following:

```
# adding a deterministic trajectory
source('Det_SI.R')
det.sol <- DetSI.dyn(500, 1, 0.5, 30)

det.t <- det.sol[,1]
det.I <- det.sol[,3]

lines(det.t, det.I, lwd=3 , col='red')
```



8.5 Exploring the effects of stochasticity

Task 67

Modify the above code to explore the effects of different initial conditions ($I_0 = 1, 5, 10, 50, 100$) on the stochastic replicates. Keep the total population size constant ($N = 500$).

Qualitatively what effect does this have on the variability between replicates?

Show: Solution on P237

Task 68

Now vary the population size ($N = 50, 500, 5000$) with $I_0 = 1$.

Qualitatively what effect does this have on the variability between replicates?

Show: Solution on P237

Task 69

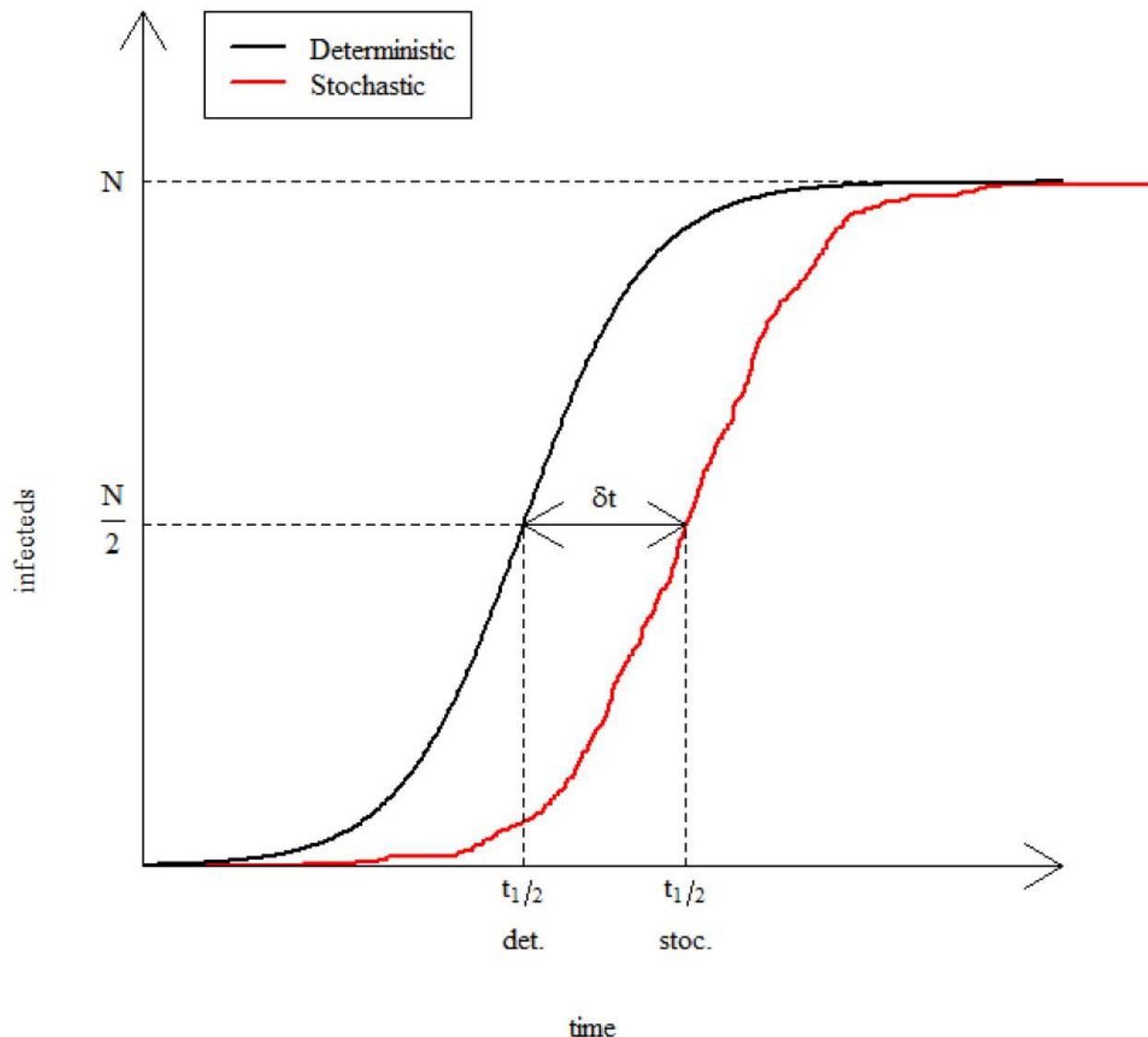
Now vary the infection rate ($\beta = 0.1, 0.5, 1$).

Qualitatively what effect does this have on the variability between replicates?

Show: Solution on P238

8.6 The delay time

Graphical exploration of model output is a good way to orientate yourself and understand the qualitative impact of changing parameters, but in practice we do need to explore various summary statistics. In this section we will compare the time it takes for a stochastic epidemic to infect half of the total population, $t_{\frac{1}{2}}$, with the time a deterministic epidemic takes. This difference is the delay time, δt (see the figure below).



We will repeat this for many stochastic replicates and plot a distribution of the delay times.

8.6.1 The deterministic half-time

The deterministic model predicts a constant value of $t_{1/2}$ given by the expression:

$$t_{1/2} = -\frac{1}{\beta} \ln \left(\frac{I_0}{N - I_0} \right)$$

(This can be derived exactly from the analytic solution to the system of differential equations for the *SI* system. This isn't going to be possible for more complex models.)

8.6.2 The stochastic half-time

We must now construct a function that calculates the value of $t_{\frac{1}{2}}$ for a set of stochastic simulations and the corresponding delay times:

Open a new script file and save as `Half_time.R` with the following content:

```
# Deterministic Halftime Function
det_halftime <- function(N, I0, beta) {

  # calculate deterministic half life
  return((-1/beta)*log(I0/(N-I0)))
}

# Stochastic Halftime Function
stoc_halftime <- function(model) {

  no_runs = n_nodes(model)
  N = sum(model@u0[,1])

  # create storage vector for stochastic half lives
  thalf <- numeric(no_runs)

  # loop over number of replicates
  for (iSim in 1:no_runs) {
    # extract the infected time series for each simulation
    output <- trajectory(out, "I", iSim)

    # find the first time when the infected population is at least half the total population
    ind <- min(which(output$I>=(N/2)))
    thalf[iSim] <- output$time[ind]
  }

  return(thalf)
}
```

The function `det_halftime` takes three arguments:

- The total host population `N`
- The initial number of infected `I0`
- The infection rate `beta`

and returns the deterministic value of the time to half-height.

The function `stoc_halftime` takes a `SimInf` model object as its input and returns a vector with the time to half height of each model simulation (to nearest day).

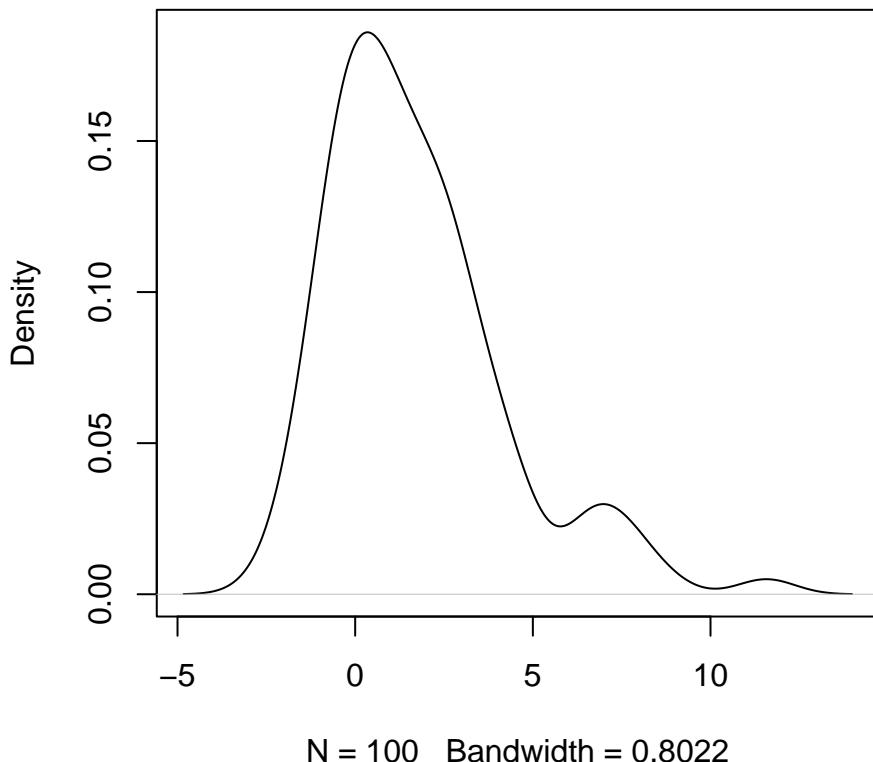
We then plot a probability density plot for these delay times using the in-built function `density`.

```
source('Half_time.R')
SImodel <- create.StocSI(500, 1, 0.5, 30, 100)
out <- run(model = SImodel)

delay_times <- stoc_halftime(out) - det_halftime(500, 1, 0.5)

plot(density(delay_times), main = '')
```

`density` produces smoothed estimates of the frequency distributions of a set of samples (essentially a smooth line through the top of a histogram). This will produce a figure similar to the one below:



Task 70

Can the distribution of delay-times be used to quantify the appropriateness of the deterministic approximation for a given set of parameters? In particular, in what limits will the deterministic model produce useful predictions for the time-scale of an *SI* epidemic? You may wish to revisit the impact of changing the host-population size N , initial number of infectives I_0 and infection rate. Produce plots to justify your reasoning and discuss them with a demonstrator.

8.7 Coding a simple *SIR* model

Using the results from the earlier section on *SI* models as a starting point we will construct a stochastic *SIR* model. The deterministic *SIR* model equations are given by

$$\frac{dS}{dt} = -\beta \frac{SI}{N}$$

$$\frac{dI}{dt} = \beta \frac{SI}{N} - \mu I$$

$$\frac{dR}{dt} = \mu I$$

Task 71

1. Using these equations construct a transition diagram for the system
2. What are the states/compartments for this system?
3. What are the transitions events and what are the rates associated with each event?
4. What is the initial state of the system (i.e. how many individuals should be in each compartment initially)?

Open a new script in RStudio.

Task 72

Using the functions developed in the previous sections as a template construct a `create.StocSIR` function that takes five arguments:

- `N`: the total population size
- `I0`: the initial number of infected individuals
- `beta`: the infection rate
- `mu`: the recovery rate
- `f_time`: the time the simulation will run for (daily steps)
- `reps`: the number of replicate simulations to run

The skeleton function below can serve as a starting point.

```
create.StocSIR <- function(N, I0, beta, mu, f_time, reps = 1) {
  initial_state <-
  compartments <-
  transitions <-
  tspan <-

  model <- mparse(transitions = transitions,
                  compartments = compartments,
                  gdata = c(beta = beta, mu = mu),
                  u0 = initial_state,
                  tspan = tspan)

  return(model)
}
```

Hint

To specify multiple transitions in the `SimInf` algorithm you will need a transitions vector with two elements to it like this:

```
transitions <- c('S -> ?? -> I', 'I -> ?? -> R')
```

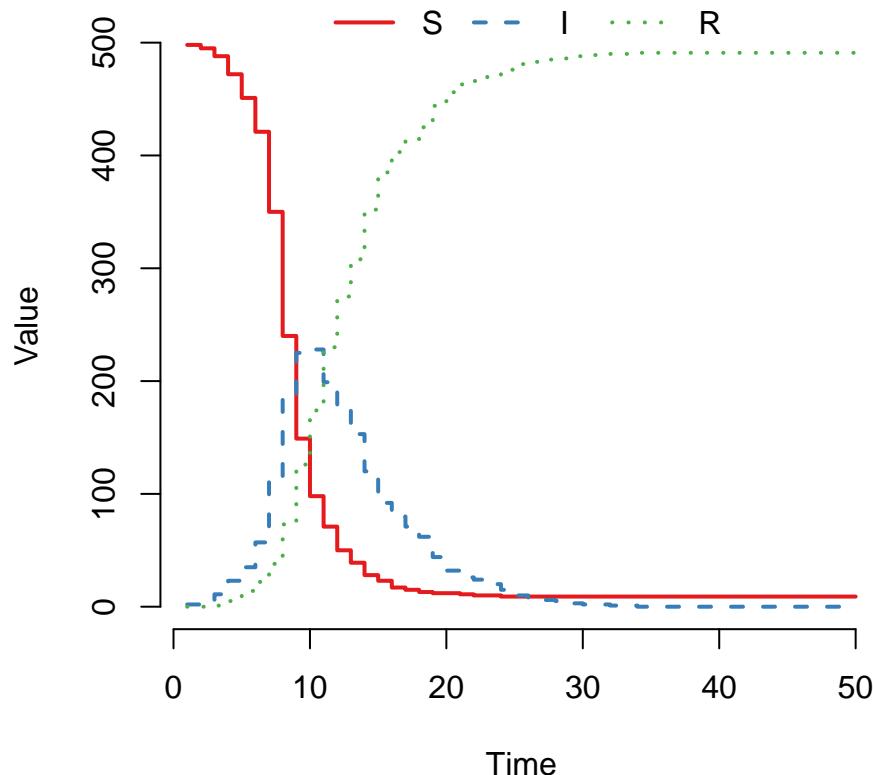
where I have deliberately replaced the relevant transition rates with `??`. You will need to replace the `??` with the correct rate equations.

Using your function run the following code:

```
SIRmodel <- create.StocSIR(500, 1, 1, 0.25, 50)
out <- run(model = SIRmodel)
plot(out)
```

Show: Solution on P239

This should produce a graph that looks like this:



However, some of you will see a very different graph. (If you do get something like this plot then try re-running the model and plotting it several times until you see something different.)

Question 73

Why might this be the case?

Show: Answer on P240

Task 74

Use this function to plot 20 replicates of the infection curves (use the code in the previous *SI* section as a guide).

Show: Solution on P240

Task 75

Load the R script called `Det_SIR.R` and adapting your code from the previous section make a plot that overlays the deterministic solution with the output of the stochastic model.

Show: Solution on P240

8.8 SIS Extension: Adding Recovery from Infection

Consider a system where individuals recover from infection with a fixed rate α and return to the susceptible class. This *SIS* system is described by the following system of equations:

$$\begin{aligned}\frac{dS}{dt} &= \alpha I - \beta \frac{SI}{N} \\ \frac{dI}{dt} &= \beta \frac{SI}{N} - \alpha I\end{aligned}$$

Task 76

1. Draw the transition diagram for this system.
2. For each transition write down the rate at which it occurs.
3. What are the parameters required for this model?
4. Write a new function `create.StocSIS` based upon the `create.StocSI` function and add to the `Stoch_SI.R` file and repeat the above analysis. Use these initial values for the parameters:
 - $\alpha = 0.1$
 - $\beta = 0.5$

Show: Solution on P241

8.9 SIS Extension: Adding host demography

Consider an *SI* model with host demography. The system has constant birth rate Λ into the susceptible class and fixed per capita mortality rate μ from both the susceptible and infectious classes and is governed by the following equations:

$$\frac{dS}{dt} = \Lambda - \beta \frac{SI}{N} - \mu S$$

$$\frac{dI}{dt} = \beta \frac{SI}{N} - \mu I$$

Task 77

Repeat the previous steps for this model and create a `create.StocSI_demo` function in `Stoch_SI.R` and then explore its dynamics.

Hint

To specify a birth (or death) event then we use the `@` symbol to represent the “empty” compartment from which births appear (or to which deaths end up). So a birth event into the susceptible compartment would have the following transition string:

`@ -> ?? -> S`

Use these initial values for the parameters:

- $\mu = 0.1$
- $\beta = 0.5$
- $\lambda = 100$

Show: Solution on P243

8.10 SIR Extension: The final epidemic size

In order to further explore the effects of stochasticity in the *SIR* model we are going to explore the how the final size of an epidemic varies between realisations.

- What is the final size of an epidemic?
- What does it mean?
- Which state variable do we need to consider in order to calculate the final epidemic size?

For stochastic simulations the final epidemic size will vary between realisations of the epidemic which will mean that we need to think about a *distribution* of final epidemic sizes.

Task 78

Write code to calculate the final epidemic size from a `SimInf` output object. Use $\beta = 1$ and $\mu = 0.25$ as before, running the outbreak for 50 days.

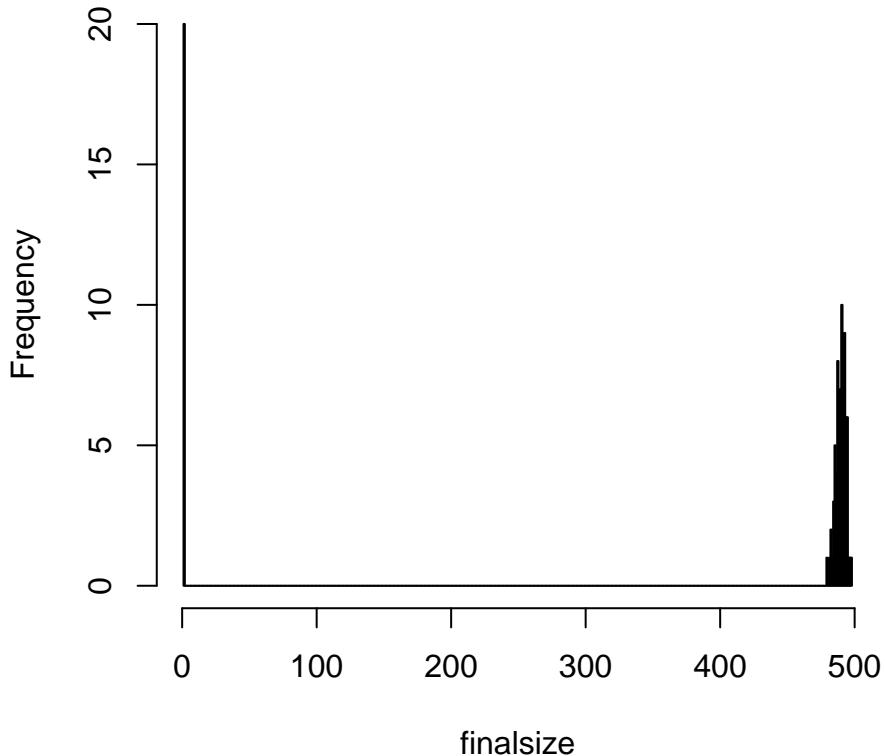
Hint

- What value should the final time for the simulations be set to (roughly)?
- As we've seen before, the function `trajectory` can be used to extract the actual values of the different compartments at each time point.
- Look at the code we used in the previous practical for calculating the halftime values for a single simulation and modify that logic accordingly.

Show: Solution on P244

Task 79

Run 100 realisations, extract the final epidemic size from each one and then produce a histogram of the final sizes. It should look something similar to the figure below:

Histogram of finalsize**Hint**

- You'll probably want to use a loop to extract the final size for each simulation
- Set the histogram to use 500 bins

Why does the histogram have a bimodal distribution?

Show: Solution on P245

Recall that the final epidemic size for the deterministic *SIR* model showed strong dependency on the value of the basic reproductive number, R_0 .

Task 80

- Write down an expression for R_0 for this system as a function of the parameters β and μ .
- Does this help explain the bimodal result?

Show: Solution on P245

For this section, we shall vary the value of R_0 by varying the transmission rate, β , whilst keeping the recovery

rate, μ , at a fixed value.

Task 81

Explore how the final size distribution varies with R_0 by producing histograms of the final size distribution as per the above figure for values of $R_0 = 20.0, 5.0, 2.0, 1.2, 1.0, 0.9, 0.5$.

Can you always identify the two modes of the distribution?

Show: Solution on P246

8.11 SIR Extension: Stochastic extinction

One of the primary differences between the deterministic and stochastic formulations of the *SIR* model is that in the stochastic formulation the epidemic becomes extinct within a finite time, the so-called extinction time. The extinction time varies between realisations and in this section we are going to explore this distribution.

Question 82

Which state variable do we need to consider in order to calculate the time to epidemic extinction and what is the value of this state variable at extinction?

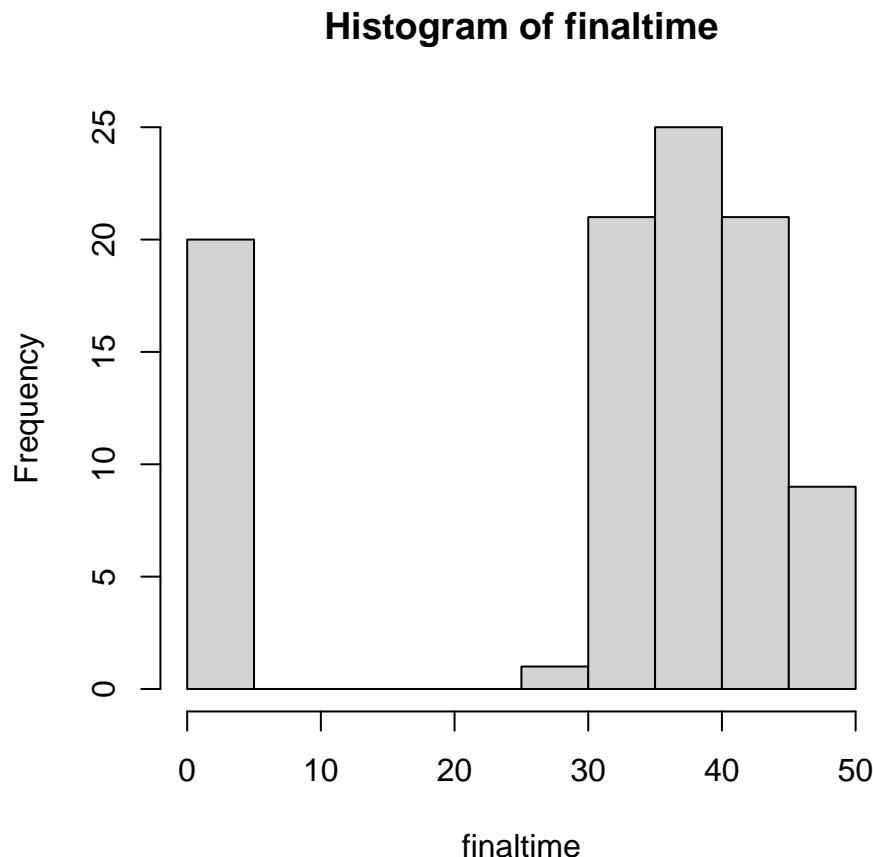
Show: Answer on P247

As with the final epidemic size, the extinction time will vary over realisations of the epidemic which will result in a distribution of extinction times, which we are now going to explore.

Task 83

Adapt your previous code to calculate the extinction time for each realisation.

- Run 100 realisations in order to produce a histogram of the time to extinction, as per the figure below:



- Explore the distribution of extinction times for several values of $R_0 = 20.0, 5.0, 2.0, 1.2, 1.0, 0.9, 0.5$.
- What do you notice about the distribution as R_0 gets close to 1?

Show: Solution on P247

Part V

Statistical modelling

Chapter 9

Estimating R_0 using catalytic models

T. J. McKinley (t.mckinley@exeter.ac.uk) and Andrew Conlan (ajkc2@cam.ac.uk)

The aim of this practical is to estimate the age-stratified force-of-infection and basic reproductive ratio from age-stratified serological data for an endemic disease.

Objectives:

1. Estimate the age-stratified force-of-infection of rubella from serological data.
2. Calculate the next generation matrix and R_0 for rubella.

We approach these two objectives using two alternative methods:

- In Section 9.1 we take the traditional two stage approach of estimating the force-of-infection first and then calculating the next generation matrix from an assumed Who-Aquires-Infection-from-Whom (WAIFW) matrix.
- In Section 9.2, we take the more elegant approach of estimating the next generation matrix directly using social contact (POLYMOD) data.

9.1 Estimating the force-of-infection and R_0 of rubella

In this practical we will estimate the force-of-infection of rubella using serological survey data from the UK provided in the book “[Modelling Infectious Disease Parameters Based on Serological and Social Contact Data](#)” by Hens et al. (2012)

You will need a copy of the data ([Rubella-UK.csv](#)), which you can download via the link. We will begin by loading and checking the contents of the rubella data table:

```
rubella <- read.csv("Rubella-UK.csv", header = TRUE)
head(rubella)
```

```
##   Age Pos Neg
## 1   1  31 175
```

```
## 2   2 30 116
## 3   3 34 134
## 4   4 57 132
## 5   5 95 124
## 6   6 104 91
```

`rubella` should be a table with 44 rows and 3 columns. `rubella$Age` is the endpoint of the age group (annual) from which `rubella$Pos` + `rubella$Neg` samples were taken, of which `rubella$Pos` were positive for rubella antibodies and `rubella$Neg` were negative.

Task 84

Estimate the probability of being susceptible (p_s) within each age category and add this as an extra column to the rubella data frame.

Show: Solution on P249

The force-of-infection (λ) is defined as the rate that susceptible individuals acquire infection per unit time. Cross-sectional surveys provide no explicit information on how the risk-of-infection varies with time (which will depend on the epidemic dynamics), but can be used to quantify how the risk of infection depends on covariates such as age. If we assume that the individuals within an age-group experience a constant average force-of-infection then the rate of change of the proportion susceptible with age will be:

$$\frac{dp}{da} = -\lambda p_0 \quad (9.1)$$

with solution:

$$p(a) = p_0 e^{-\lambda a} \quad (9.2)$$

where p_0 is the proportion susceptible at the beginning of the age group.

The proportion susceptible at the end of an age group, p_1 , will then be given by:

$$p_1 = p_0 e^{-\lambda \Delta a} \quad (9.3)$$

where Δa is the width of the age group.

Task 85

Take logs of both sides of (9.3) and solve for λ to get a first estimator for the force of the infection in a discrete age group as a function of the proportion susceptible at the beginning and end of the age group.

Show: Hint on P265

Show: Solution on P250

Task 86

Use the equation derived above to estimate the force-of-infection λ_i within each annual age cohort ($i = 1, \dots, N_A$) and add as another column to the `rubella` data frame (e.g. `rubella$lambda`).

Show: Hints on P266

Show: Solution on P250

To check your calculations and visualise the data generate the following plots:

```
## size is a vector to scale the size of points according to the
## sample size within that age group
size <- (rubella$Pos + rubella$Neg) / max(rubella$Pos + rubella$Neg)

## now produce plots
plot(rubella$Age, rubella$p_s, pch = 19, cex = size^0.75)
plot(rubella$Age, rubella$lambda, pch = 19, cex = size^0.75)
```

Question 87

Are these “first” estimates of the force-of-infection biologically plausible (Figs 9.1 and 9.2)? If not, how has this transpired and how can we improve the estimates?

Show: Answer on P250

Dynamic age-structured epidemic models typically assume that the force-of-infection is constant within coarse age-groups, but can differ between age-groups. For such a coarse-grained models (with age groups of width a , indexed by i as before) we can calculate the piecewise probability of remaining susceptible as introduced in lectures:

$$P_s(a) = \begin{cases} e^{-\lambda_1 a} & \text{for } a_0 < a < a_1 \\ P_s(a_i) e^{-\lambda_{i+1}(a-a_i)} & \text{for } a_i < a < a_{i+1}, \end{cases} \quad (9.4)$$

where λ_i is the age-specific force-of-infection in group i .

The function `piecewise_s(a, foi)` provided below returns this age-stratified probability of being susceptible at the end of each discrete age groups, specified by the vector of ages `a` for a given vector of force-of-infections (`lambda`).

```
## Calculate proportion of population that remains susceptible
## by age a for population with force of infection (foi)
##
```

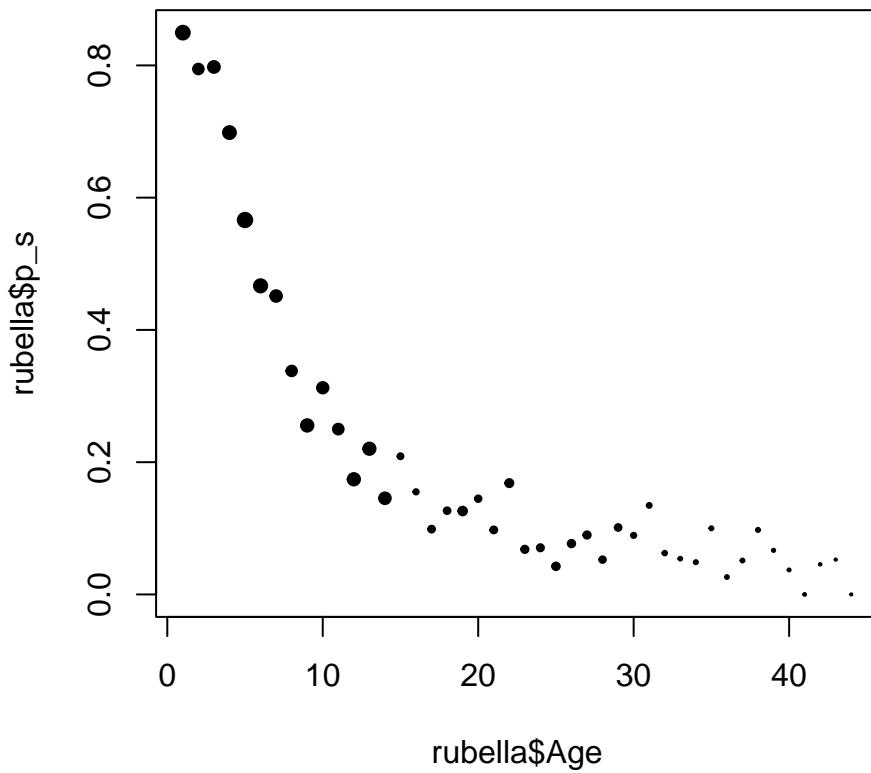


Figure 9.1: Proportion susceptible with respect to age

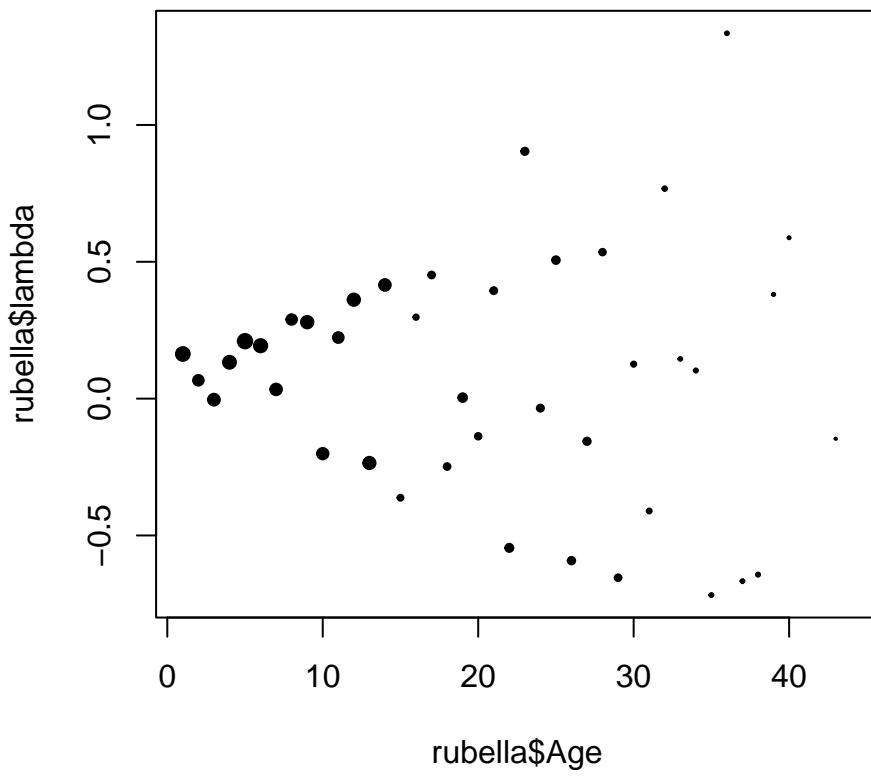


Figure 9.2: First estimate of force-of-infection of rubella

```

## Arguments:
## a: vector of ages to calculate susceptible proportion
## lambda: vector of length equal to the maximum assumed age in the population
## with a specified force of infection for each annual cohort.
piecewise_s <- function(a, lambda) {

  ## extract information on age-classes
  max_age <- length(lambda)
  breaks <- 0:max_age
  widths <- rep(1, max_age)

  ## set up output vector
  s <- numeric(length(a))

  ## loop over age-classes
  for(i in 1:length(a)) {
    if(a[i] == 0) {
      s[i] <- 1
    } else {
      ## which age class does current value of a lie within
      age_class <- max(which(a[i] > breaks))
      sg <- 1
      if(age_class > 1) {
        ## iterate over previous age groups calculate susceptibles
        ## at beginning of age group
        for(j in 1:(age_class - 1)) {
          sg <- sg * exp(-lambda[j] * widths[j])
        }
      }
      ## calculate susceptible proportion at age a within final
      ## age group
      s[i] <- sg * exp(-lambda[age_class] * (a[i] - breaks[age_class]))
    }
  }
  ## return vector of probabilities
  return(s)
}

```

The function `FOI_loglik(x, foi_key, sero_data)` defined below uses `piecewise_s()` to calculate the (binomial) likelihood of observing a given number of positive/negative samples for each age-group given in the table `sero_data`, for a piecewise continuous force-of-infection defined by vectors `x` and `foi_key`.

As individuals can only take two possible values, positive or negative, the likelihood of a given force-of-infection (FOI) can be written in terms of a binomial trial depending on the number of positive samples in each group (X_a), number of samples (N_a) and the predicted probability of infection within that age group ($p_a(K)$). Hence the log-likelihood can be written as:

$$l(K) = \sum_a \log \binom{N_a}{X_a} + X_a \log [p_a(K)] + (N_a - X_a) \log (1 - p_a(K)). \quad (9.5)$$

We can code this in R using the following function:

```
## Calculate (Binomial) likelihood of observing given number of
## positive/negative samples (provided in table sero_data) for each
## age group and specified by a piecewise continuous force of
## infection model defined by foi_key
##
## Arguments:
##   x: vector of unique force-of-infection values
##   foi_key: pattern vector of length equal to the maximum assumed age
##             in population (or data set) that specifies which unique value of x
##             corresponds to each annual cohort in population.
##   sero_data: data frame with columns
##             sero_data$Pos: positive samples
##             sero_data$Neg: negative samples

FOI_loglik <- function(x, foi_key, sero_data) {

  ## set up age classes
  breaks <- 0:length(foi_key)

  if(any(x <= 0)) {
    ## if any proposed lambdas are negative, then
    ## return zero likelihood (so log-likelihood = -Inf)
    return(-Inf)
  } else {
    ## extract relevant FOI for each age-cohort
    lambda <- x[foi_key]

    ## calculate log-likelihood
    loglik <- 0
    for(i in 1:nrow(sero_data)) {
      ## which age class does observation lie in
      age_class <- max(which(sero_data$Age[i] > breaks))

      ## calculate proportion of susceptibles at given age class
      sg <- piecewise_s(breaks[age_class], lambda)

      ## log-likelihood contribution
      loglik <- loglik + dbinom(
        sero_data$Pos[i],
        sero_data$Pos[i] + sero_data$Neg[i],
        prob = 1 - sg * exp(-lambda[i] * (sero_data$Age[i] - breaks[age_class])),
        log = TRUE
      )
    }
    ## return log-likelihood
    return(loglik)
  }
}
```

We will assume there are two mixing groups and two unique forces-of-infection, one corresponding to children < 15 years and one for adults 15–45. We wish to estimate two parameters, λ_1 and λ_2 , and we specify which parameter corresponds to which age cohort using a vector `two_groups` defined as:

```
two_groups <- (1:max(rubella$Age) >= 15) + 1
two_groups
```

```
## [1] 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [39] 2 2 2 2 2 2
```

We can obtain the maximum-likelihood estimate for this model by optimising the value of `FOI_loglik()` using the built-in function `optim()`.

Show: The `optim()` function on P266

```

## calculate MLE
twogroup_mle <- optim(
  c(0.07, 0.07),                      ## set initial values for lambda1 and lambda2
  FOI_loglik,                           ## log-likelihood function
  foi_key = two_groups,                 ## foi_key
  sero_data = rubella,                  ## data
  control = list(fnscale = -1)          ## maximise (rather than minimise) function
)
twogroup_mle

## $par
## [1] 0.12423142 0.05500056
##
## $value
## [1] -110.846
##
## $counts
## function gradient
##       83      NA
##
## $convergence
## [1] 0
##
## $message
## NULL

```

The `convergence = 0` part suggests that the optimisation has converged (in practice one might want to run `optim()` multiple times from different initial conditions to check that it's not converged to a local mode). We can check the fit of the model by plotting the predicted values against the data:

```

ages <- seq(0, 44, by = 0.1)
plot(
  ages,
  piecewise_s(ages, twogroup_mle$par[two_groups]),
  type = 'l',
  xlab = "Age",
  ylab = "Proportion susceptible"
)
points(rubella$Age, rubella$p_s, pch = 19, cex = size^0.75)

```

Task 88

Can you suggest, and estimate, a model to improve the fit to the data? How can you compare the fit of the two models?

Show: Solution on P250

9.1.1 Bayesian estimation

We can also estimate the posterior distributions for the two force-of-infections within a Bayesian framework using MCMC. Here we introduce and use one of the standard MCMC libraries in R (`MCMCpack`) that implements a generic MCMC method.

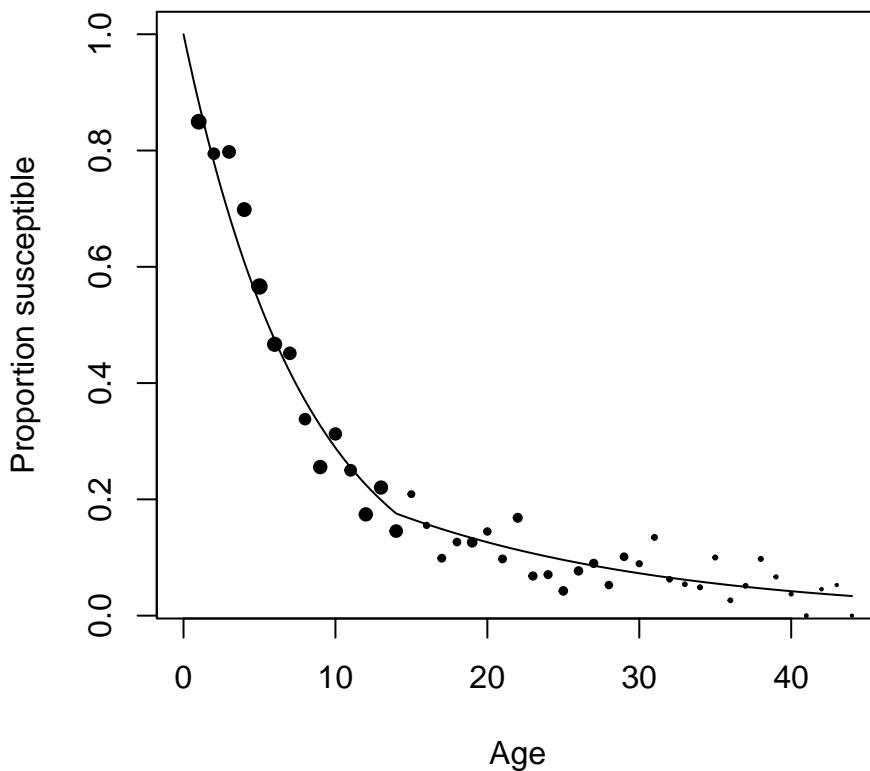


Figure 9.3: Predicted susceptible profile of two-group FOI model against data

Show: A note on priors on P266

```
## load library
library(MCMCpack)

## fit model
twogroup_post <- MCMCmetrop1R(
  FOI_loglik, ## log-likelihood
  theta.init = twogroup_mle$par, ## initial values
  mcmc = 10000, ## number of iterations
  burnin = 500, ## burn-in
  tune = 1, ## proposal parameter
  verbose = 500, ## print progress every 500 it
  logfun = TRUE, ## return log-density
  foi_key = two_groups, ## foi key
  sero_data = rubella ## data
)
```

Task 89

Check the convergence of the MCMC and comment on the weight of evidence that the force-of-infection for rubella is different in the two age groups.

Show: Hint on P266

Show: Solution on P252

9.2 Next-generation matrix approaches

If the force-of-infection was independent of age we could estimate R_0 straightforwardly as the ratio of the force-of-infection and life expectancy. However, when the force-of-infection varies with age we need to first use the force-of-infection to calculate the transmission parameters within and between each age-group. Here we work through the traditional method for achieving this where social contact data is not available and assumptions must be made about Who Acquires Infection from Whom (WAIFW).

We can derive an equation that links the force-of-infection and the elements of the next generation matrix k_{ij} . (The details of the derivation are a little messy and will be omitted here, see Anderson and May (1991); Hens et al. (2012) for details.)

$$\begin{pmatrix} \beta_{11} & \beta_{12} \\ \beta_{21} & \beta_{22} \end{pmatrix} \begin{pmatrix} \psi_1 \\ \psi_2 \end{pmatrix} = \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} \quad (9.6)$$

ψ is a function derived from the force-of-infection that, loosely, corresponds to the average proportion of individuals infectious within each age group at any time. The function `Psi(lambda, age_widths, TI, u)` defined below returns the vector ψ for a specified force-of-infection `lambda`, for coarse age groups of widths `age_widths`, and infectious period of length `TI`, and the mortality rate for each age group `u`.

```
Psi <- function(lambda, age_widths, u, TI) {

  ## set up auxiliary vectors
  psi <- numeric(length(lambda))
  Psi <- numeric(length(lambda))

  ## calculate elements of psi-vector
  psi <- lambda * age_widths
  psi <- cumsum(psi)
  Psi[1] <- 1 - exp(-psi[1])
  for(a in 2:length(lambda)) {
    Psi[a] <- exp(-psi[a - 1]) - exp(-psi[a])
  }

  ## return psi-vector
  return((u / (1 / TI)) * Psi)
}
```

Equation (9.6) defines two simultaneous equations for two known values: (λ_1, λ_2) .

We can only proceed to solve these equations by restricting the values of k_{ij} to the number of estimated forces-of-infection. Different assumptions about the structure of this matrix correspond to very different estimates of R_0 . Greenhalgh and Dietz (1994) demonstrated that the feasible **lower bound** on R_0 for these data corresponds to the situation where infants are responsible for all transmission within the population:

$$\text{infant} = \begin{pmatrix} k_1 & 0 \\ k_2 & 0 \end{pmatrix}. \quad (9.7)$$

The **upper bound** corresponds to the diagonal matrix where mixing occurs purely within your own age group:

$$\text{diagonal} = \begin{pmatrix} k_1 & 0 \\ 0 & k_2 \end{pmatrix}. \quad (9.8)$$

Task 90

By substituting these matrices into equation (9.6) and carrying out the matrix multiplication, find expressions for k_1 and k_2 in terms of $(\lambda_1, \lambda_2, \psi_1, \psi_2)$.

Show: Solution on P254

Task 91

Use your new expressions to calculate the values of k for the infant mixing and assortative mixing assumptions, and the corresponding lower and upper bounds for R_0 . Use the maximum likelihood estimates of λ_1 and λ_2 . Assume that the mortality rate $u = 20/1000$ and that $T_I = 5/365$. Assume the total population size is $n = 1$.

Show: Hint on P267

Show: Solution on P254

Task 92

Calculate the posterior distributions for R_0 for the two-group model under the same conditions as the previous task. Plot the posterior densities.

Show: Hint on P267

Show: Solution on P255

9.3 Estimating the next generation matrix from rubella serology

For human infectious diseases at least, the POLYMOD social contact data has revolutionised how we interpret serological data. In this section we estimate the next generation matrix for rubella using the POLYMOD mixing data. We also introduce a more modern, and convenient, method where we directly estimate the next generation matrix from age-stratified serological data.

For a given next generation matrix \mathbf{K} we can calculate the resulting force-of-infection using:

$$\lambda_i = \sum_j k_{ij} [P_s(a_{i+1}) - P_s(a_i)] \quad (9.9)$$

Although we cannot write down a closed form solution for K_{ij} , for any given next generation matrix \mathbf{K} , we can iteratively solve equations (9.4) and (9.9) to obtain the susceptibility curve $P_s(a)$ and force-of-infection λ_i that correspond to particular values of \mathbf{K} . By exploring different values (and forms) of \mathbf{K} we can then estimate what next generation matrix is most consistent with a given susceptibility curve.

This procedure is straightforward, but a bit fiddly, so once again we have provided you with helper functions below to do this, adapted from the book by Hens et al. (2012).

`NextGen_foi(K)` returns the age-stratified force-of-infection corresponding to a specified next generation matrix \mathbf{K} (Equation (9.9) above). The number of age groups is taken from the dimensions (number of rows/columns) of \mathbf{K} .

`NextGen_loglik(x, matrixfunction, sero_data)` is analogous to `FOI_loglik()` and returns the likelihood of observing a given number of positive/negative samples for each age-group and a specified next generation matrix with parameters `x` and structure defined by a user specified function `matrixfunction()`.

Note that instead of estimating the scaling parameter $y = e^x$ directly, we estimate $x = e^y$. This parameter transformation doesn't change our basic assumption that transmission is directly proportional to the reported POLYMOD contacts, but improves the numerical accuracy of our estimation. Such tricks often improve the convergence of numerical procedures, especially MCMC methods, but the appropriate choice can be somewhat of a black art.

By writing `NextGen_loglik()` to depend on a function as well as a set of parameters we can easily fit alternative models that assume different patterns of mixing in the population. For example we can specify a homogeneous mixing model by the following function:

```
## function that returns a homogeneous mixing next
## generation matrix with all elements equal to x
HomogeneousMatrix <- function(x) {
  mat <- matrix(x, 86, 86)
  return(mat)
}
```

where we assume there are 86 age cohorts in the population (for consistency with POLYMOD matrix).

The helper functions are defined below:

```

## calculate force of infection for a given next generation matrix K
##
## annual age cohorts are assumed with maximum assumed age given by
## dimensions of K
##
## Arguments:
## K - Next Generation Matrix
NextGen_foi <- function(K) {
  ## set up age-class widths
  widths <- rep(1, nrow(K))
  ## number of age classes
  num_class <- nrow(K)
  ## piecewise FOI by age-class
  lambda <- numeric(num_class)
  ## susceptible fraction at END of age class
  s <- rev(sort(runif(num_class)))
  oldlambda <- lambda
  tolerance <- 1
  iterations <- 1
  while ((tolerance > 1e-10) & (iterations < 2000)) {
    oldlambda <- lambda
    for(i in 1:num_class) {
      lambda[i] <- (K[i, 1] * (1 - s[1]) +
                     sum(K[i, 2:num_class] *
                         (s[1:(num_class - 1)] - s[2:num_class])))
    }
    s[1] <- exp(-lambda[1] * widths[1])
    for(i in 2:(num_class)) {
      s[i] <- s[i - 1] * exp(-lambda[i] * widths[i])
    }
    tolerance <- sum((lambda - oldlambda)^2)
    iterations <- iterations + 1
  }
  return(lambda)
}

```

```

## calculate (Binomial) likelihood of observing given number of
## positive/negative samples
## (provided in table sero_data) for each age group and specified
## next generation matrix K = matrixfunction(x)
##
## Arguments:
## x: vector of unique force of infection values
## foi_key: pattern vector of length equal to the maximum assumed age
##           in population (or data set) that specifies which unique value of
##           x corresponds to each annual cohort in population.
## sero_data: data frame with columns
## sero_data$Pos: positive samples
## sero_data$Neg: negative samples
NextGen_loglik <- function(x, matrixfunction, sero_data) {

  ## set up auxiary vectors
  x <- exp(x)
  K <- matrixfunction(x)
  breaks <- 0:nrow(K)
  widths <- rep(1, nrow(K))

  if(any(x <= 0)) {
    ## if any rates < 0 then
    ## return log-likelihood = -Inf
    return(-Inf)
  } else {
    ## calculate fois
    lambda <- NextGen_foi(K)

    ## calculate log-likelihoods
    loglik <- 0.0
    for(i in 1:nrow(sero_data)) {
      ## which age class does observation lie in
      age_class <- max(which(sero_data$Age[i] > breaks))
      ## proportion of susceptibles at beginning of age class
      sg <- piecewise_s(breaks[age_class], lambda)
      loglik <- loglik + dbinom(
        sero_data$Pos[i],
        sero_data$Pos[i] + sero_data$Neg[i],
        prob = 1 - sg * exp(-lambda[age_class] *
          (sero_data$Age[i] - breaks[age_class])),
        log = TRUE
      )
    }
    ## return log-likelihood
    return(loglik)
  }
}

```

Show: The optimise() function on P267

We can now obtain the maximum-likelihood estimate for the homogeneous mixing matrix by optimising the value of `NextGen_loglik()`:

```
homogeneous_mle <- optimise(
  NextGen_loglik,
  log(c(0.01, 1)),
  matrixfunction = HomogeneousMatrix,
  sero_data = rubella,
  maximum = TRUE
)
homogeneous_mle
```

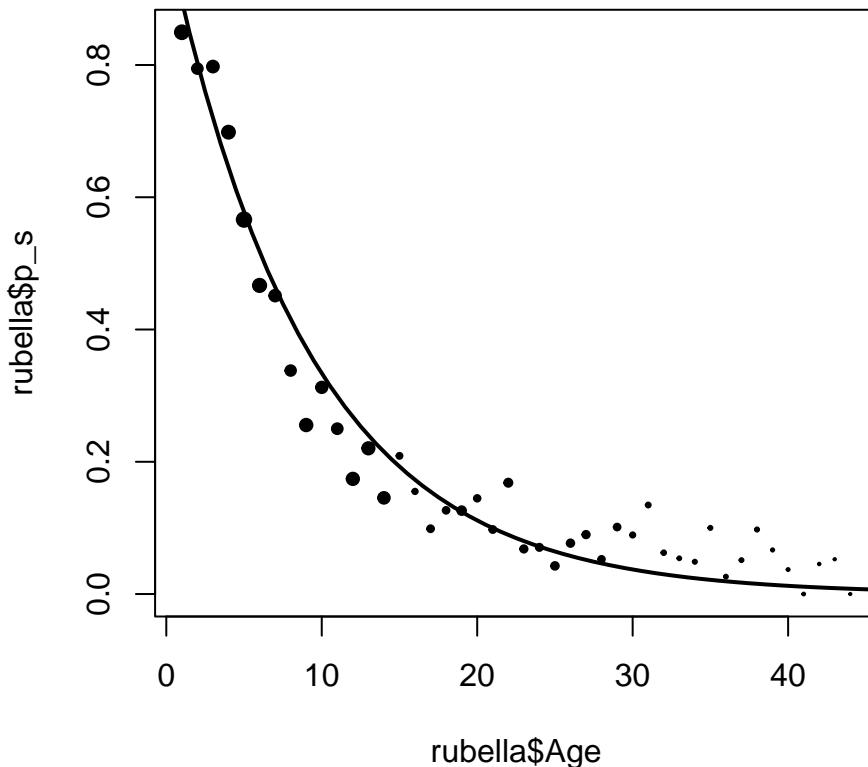
```
## $maximum
## [1] -2.209086
##
## $objective
## [1] -135.6114
```

Then to calculate the force-of-infection corresponding to our maximum likelihood fit:

```
homogeneous_foi <- NextGen_foi(
  HomogeneousMatrix(exp(homogeneous_mle$maximum))
)
```

You can check the fit of the model by using the `piecewise_s()` function to predict the proportion positive for each age group and plot against the original data:

```
plot(rubella$Age, rubella$p_s, pch = 19, cex = size^0.75)
lines(
  seq(0.5, 75, 1),
  piecewise_s(
    seq(0.5, 75, 1),
    homogeneous_foi
  ),
  lwd = 2
)
```



The overall fit is actually pretty good. However if we look closely the model does systematically overestimate the proportion infected in 10–20 year olds and underestimates in the 20–44 age groups, as we would expect from our analysis in the earlier section.

We could generalise our mixing matrix to include additional parameters to describe this heterogeneity as before. Or we can test whether social contact patterns provide a better description of the observed force-of-infection. Here we assume that infectious contacts are distributed proportionally to a POLYMOD mixing matrix.

Download the [POLYMOD.csv](#) file to your working directory and load into R:

```
## load (smoothed symmetrical) POLYMOD matrix
POLYMOD <- as.matrix(read.csv("POLYMOD.csv", header = FALSE))
```

Now we write a function called `POLYMODMatrix()` that takes a single parameter `x` and returns a scaled version of the POLYMOD matrix (M_{ij}):

```
## generates a scaled POLYMODMatrix with transmission
## parameter defined by x
POLYMODMatrix <- function(x) {
  return(x * POLYMOD)
}
```

Task 93

Follow the same steps as before to find the maximum likelihood estimate of the next generation matrix for the POLYMOD assumption and create a plot comparing the estimated force-of-infection and model fit for both models.

Show: Solution on P257

Finally, calculate the value of R_0 for the two mixing assumptions:

```
max(Re(eigen(HomogeneousMatrix(exp(homogeneous_mle$maximum)))$values))
```

```
## [1] 9.442879
```

```
max(Re(eigen(POLYMODMatrix(exp(POLYMOD_mle$maximum)))$values))
```

```
## [1] 7.777768
```

Question 94

Does the POLYMOD mixing provide a better fit to the rubella serological data?

Show: Answer on P258

Question 95

How do the predictions of the POLYMOD mixing model differ from homogeneous mixing?

Show: Answer on P258

Chapter 10

Particle MCMC

T. J. McKinley (t.mckinley@exeter.ac.uk)

Here we use a simple model to illustrate how we can use particle Markov chain Monte Carlo (PMCMC) routine ([Andrieu, Doucet, and Holenstein 2010](#)) with a bootstrap particle filter ([Gordon, Salmond, and Smith 1993](#)) as a way of fitting compartmental models to partially observed data, as described in the lectures.

Note that PMCMC is extremely computationally intensive, and the only real way to make it tractable for many problems is to code both the simulation code and the MCMC code in a low-level language such as C/C++. We could use e.g. the `pomp` package to fit this model using the same PMCMC routine described here, but the syntax is different to the `SimInf` style. So instead we will use the `SimBIID` package, which provides a function `PMCMC()` that runs this algorithm. You must pass a `SimBIID_model` object (see [here](#)) to this function, and then it will automatically compile in the correct manner.

Please note that an alternative frequentist approach, using maximum likelihood via iterated filtering (MIF) framework of Ionides, Bretó, and King ([2006](#)), and implemented in the `pomp` package, can be found [here](#).

10.1 Case Study

To illustrate some of these ideas we will use a case study of influenza in a boarding school. These data are from a paper in the BMJ in 1978 ([Anonymous 1978](#)) and provided in the `outbreaks` R package. We use a simple $SIRR_1$ model with two removal classes, R and R_1 . We will assume that the bed-rest counts in the data correspond to the number of individuals in the R class, and will ignore the other time-series for the time being. This has been used various times in the literature, including Murray ([2003](#)), de Vries et al. ([2006](#)) and in some of the `pomp` tutorials. The stochastic model we will use has event probabilities:

$$\begin{aligned} P[S_{t+\delta t} = S_t - 1, I_{t+\delta t} = I_t + 1] &\approx \beta SI/N \\ P[I_{t+\delta t} = I_t - 1, R_{t+\delta t} = R_t + 1] &\approx \gamma I \\ P[R_{t+\delta t} = R_t - 1, R_{1,t+\delta t} = R_{1,t} + 1] &\approx \gamma_1 R \end{aligned}$$

The initial population size is 763 pupils, and we assume an initial introduction of infection of a single individual at day 0.

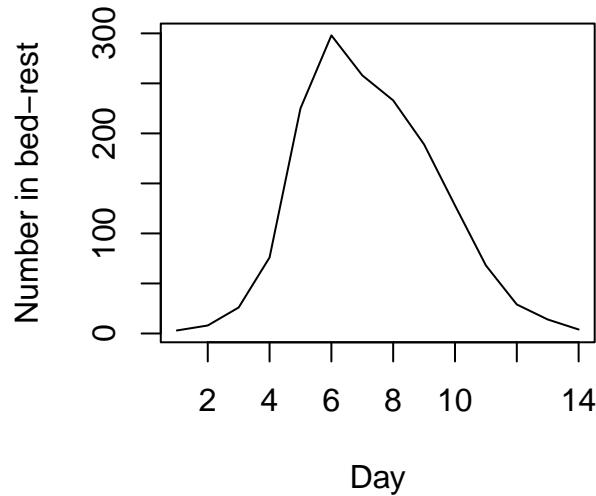
First, we load the data and the `SimBIID` package:

```
## load libraries
library(outbreaks)
```

```
library(SimBIID)

## set up data
flu <- influenza_england_1978_school
flu$day <- 1:nrow(flu)

## plot data
plot(flu$day, flu$in_bed, xlab = "Day", ylab = "Number in bed-rest", type = "l")
```



10.2 Arguments for PMCMC() function

If you look at the help file for the `PMCMC()` function (e.g. `?PMCMC`), you will see the main arguments to the `PMCMC()` function , which are summarised below:

- `x`: a `data.frame` containing time series count data, with the first column called `t`, followed by columns of time-series counts. The time-series counts columns must be in the order of the `counts` object in the `func` function (see below).
- `priors`: a `data.frame` containing columns: `parnames`, `dist`, `p1` and `p2`, with number of rows equal to the number of parameters. The column `parname` simply gives names to each parameter for plotting and summarising. Each entry in the `dist` column must contain one of `c("unif", "norm", "gamma")`, and the corresponding `p1` and `p2` entries relate to the hyperparameters (lower and upper bounds in the uniform case; mean and standard deviation in the normal case; and shape and rate in the gamma case).
- `func`: a `SimBIID_model` object (which can be created using `mparseRcpp()`). This must have a stochastic observation process specified—see Section 10.2.2.
- `u`: a named vector of initial states.
- `npart`: an integer specifying the number of particles for the bootstrap particle filter.
- `iniPars`: a named vector of initial values for the parameters of the model. If left unspecified, then these are sampled from the prior distribution(s).
- `niter`: an integer specifying the number of iterations to run the MCMC.

10.2.1 Data

The `x` argument that we will pass to the `PMCMC()` function will be a `data.frame` with the first column corresponding to `t` and the second corresponding to the *observed R* curve. Here we set up a `data.frame` called `flu_dat` that is in the correct format:

```
## set up data to pass to PMCMC
flu_dat <- data.frame(t = flu$day, Robs = flu$in_bed)
head(flu_dat)
```

```
##   t Robs
## 1 1    3
## 2 2    8
## 3 3   26
## 4 4   76
## 5 5  225
## 6 6  298
```

10.2.2 Observation process

When we specify our simulation model using `mparseRcpp()`, we also need to specify a **stochastic** observation process. This is passed as an argument called `obsProcess` to the `mparseRcpp()` function. This argument must be a `data.frame`, with columns in the order: `dataNames`, `dist`, `p1`, `p2`.

- `dataNames` is a character denoting the observed data (must match a column in the `x` data frame—see Section 10.2.1);
- `dist` is a character specifying the distribution of the observation process (must be one of "unif", "pois" or "binom" at the current time);
- `p1` is the first parameter (the lower bound in the case of "unif", the rate in the case of "pois", or the size in the case of "binom");
- `p2` is the second parameter (the upper bound in the case of "unif", `NA` in the case of "pois", and `prob` in the case of "binom").

Here we will place a Poisson observation process around the R curve, such that:

$$R_t \sim \text{Po}(R'_t + 10^{-6}),$$

where R_t is the **observed** R count at time t , R'_t is the simulated count. We add a small constant (10^{-6} here), which is important to prevent numerical errors, since the simulated counts R'_t could be zero, which would result in the Poisson rate parameter being zero, which violates the conditions of the Poisson distribution and would thus produce non-finite likelihood estimates. The addition of a small constant prevents this from happening.

Show: Note on Poisson assumptions on P267

The observed data, R_t , is coded as the `Rob`s column in the `flu_dat` data frame—see Section 10.2.1. To set up the observation process defined above, we define a `data.frame` as follows:

```
## set up observation process
obs <- data.frame(
  dataNames = "Rob",
  dist = "pois",
  p1 = "R + 1e-5",
  p2 = NA,
  stringsAsFactors = FALSE
)
obs
```

```
##   dataNames dist      p1 p2
## 1       Robs  pois R + 1e-5 NA
```

10.2.3 Setting up the model

Note

We do not dwell on how to use `SimBIIID` to specify simulation models here. If you are interested, then please see the extended documentation available at:

https://tjmckinley.github.io/SimBIIID_tutorial/set-up-simple-simulation-model.html

Hopefully the code below will look similar to `SimInf`, and you will be able to see how the model is set-up.

To specify the above simulation model:

```
## set up model
transitions <- c(
  "S -> beta * S * I / (S + I + R + R1) -> I",
  "I -> gamma * I -> R",
  "R -> gamma1 * R -> R1"
)
compartments <- c("S", "I", "R", "R1")
pars <- c("beta", "gamma", "gamma1")
model <- mparseRcpp(
  transitions = transitions,
  compartments = compartments,
  pars = pars,
  obsProcess = obs
)
```

Note

`SimBIIID` has an option to pre-compile models, but we do **not** do this here. The `PMCMC()` function will do this for us. This is because we need to compile as an object to run from C rather than R, so the `PMCMC()` function deals with this automatically.

10.3 Running the PMCMC algorithm

Now we run the PMCMC algorithm for 5,000 iterations, using 25 particles. We pass the a set of initial states, and use $U(0, 5)$ priors for each of the three parameters. We print summaries to the screen every 1,000 iterations (`nprintsum = 1000`):

```
## set priors
priors <- data.frame(
  parnames = c("beta", "gamma", "gamma1"),
  dist = rep("unif", 3),
  stringsAsFactors = FALSE)
priors$p1 <- c(0, 0, 0)
priors$p2 <- c(5, 5, 5)

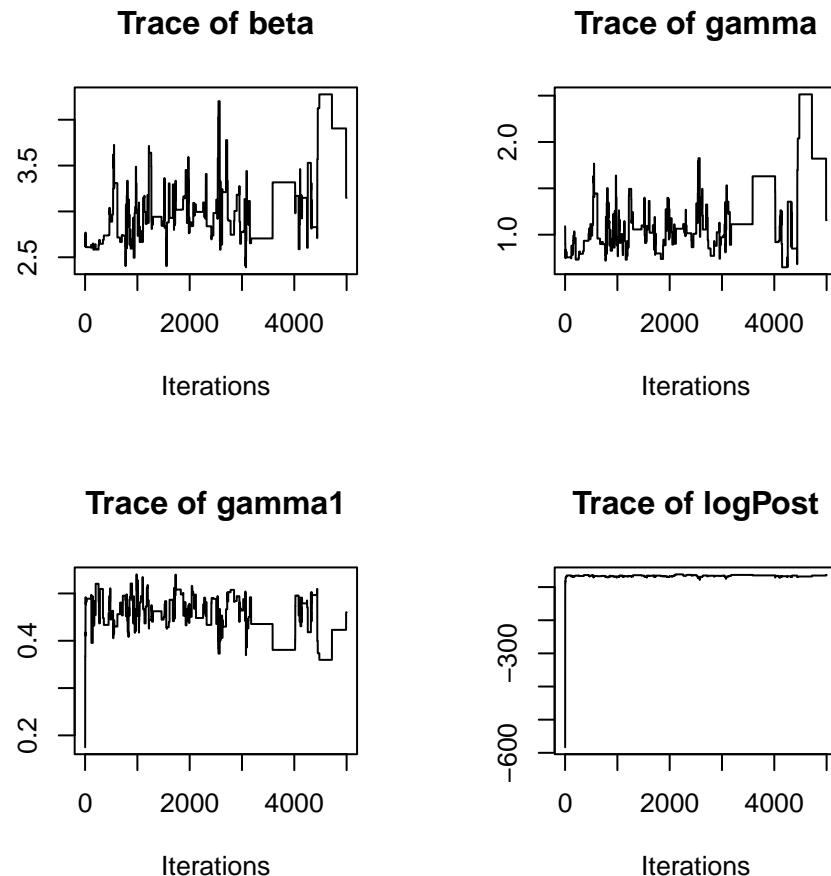
## define initial states
iniStates <- c(S = 762, I = 1, R = 0, R1 = 0)
```

```
## run PMCMC algorithm
post <- PMCMC(
  x = flu_dat,
  priors = priors,
  func = model,
  u = iniStates,
  npart = 25,
  niter = 5000,
  nprintsum = 1000
)
```

```
## Number of iterations: 5000
## Number of particles: 25
## Mixing proportion for proposal: 0.05
## Start adaptive proposal at iteration: 100
##
## Number of parameters: 3
##
## Priors:
## beta ~ U(lower = 0, upper = 5)
## gamma ~ U(lower = 0, upper = 5)
## gamma1 ~ U(lower = 0, upper = 5)
##
## Number of classes: 4
##
## Initial states:
## state[0] = 762
## state[1] = 1
## state[2] = 0
## state[3] = 0
##
## Initialising system...
## Initialisation complete!
##
## Initial parameter values:
##
```

```
## beta = 2.73907
## gamma = 1.0908
## gamma1 = 0.17482
##
## Starting runs...
## i = 1000 acc = 0.07 time = 5.98 secs
## i = 2000 acc = 0.06 time = 6.07 secs
## i = 3000 acc = 0.05 time = 5.96 secs
## i = 4000 acc = 0.02 time = 5.38 secs
## i = 5000 acc = 0.03 time = 4.97 secs
## Final time = 28.36 secs
```

```
## plot MCMC traces
plot(post, "trace")
```

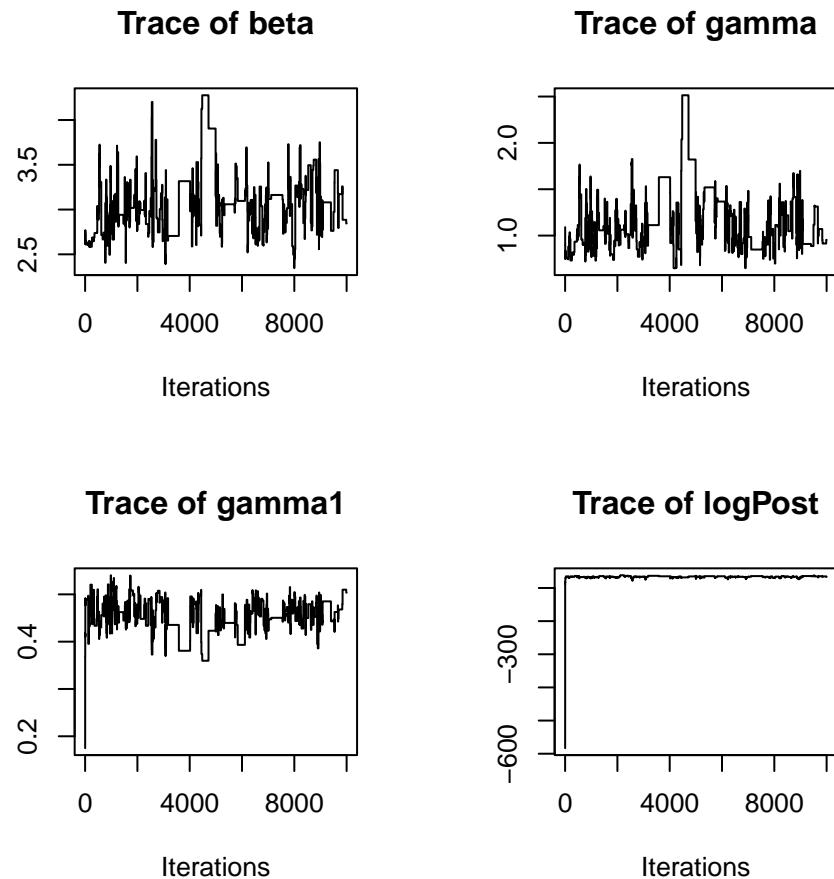


We can see that the chain looks like it's converging towards a stationary distribution, but let's run it for a bit longer. We can do this simply by passing our current PMCMC object back into the PMCMC() function:

```
post <- PMCMC(post, niter = 5000, nprintsum = 1000)
```

(I've suppressed the output for brevity here...)

```
plot(post, "trace")
```



10.4 Optimising the number of particles

The mixing of the chain and the speed of convergence is related to the number of particles (amongst other things). There is no strong consensus, but a rule-of-thumb is to try to choose the number of particles such that the variance of the log-likelihood estimate at a suitable set of parameters θ' is between 1–3. Clearly the larger the number of particles, the higher the computational burden, so in practice the additional computational burden of the simulations must be balanced against the improved mixing and faster convergence. This is tricky, so instead here we take a simpler approach.

Firstly we run the chain for a fixed number of particles until it looks like the chain has converged. Then we choose a set of parameter values θ' (the posterior medians here). We then generate 500 estimates of the log-likelihood for a range of different numbers of particles, from which we can calculate the variance of these estimates. We then choose the smallest number of particles with a variance of the log-likelihood of less than 3.

Hence, from the training runs above we can remove some burn-in iterations, and extract the posterior medians:

```
postMed <- window(post, start = 2000)
postMed <- as.matrix(postMed$pars)
postMed <- apply(postMed, 2, median)
postMed <- postMed[-length(postMed)]
postMed
```

```
##      beta      gamma     gamma1
```

```
## 3.0804375 1.0818692 0.4504811
```

We can produce 500 estimates of the log-likelihood by setting the `fixpars = TRUE` argument to the `PMCMC()` function, passing in the `postMed` estimates above.

```
flu_train <- PMCMC(
  x = flu_dat,
  priors = priors,
  func = model,
  u = iniStates,
  npart = 25,
  iniPars = postMed,
  niter = 500,
  fixpars = TRUE
)
```

This produces a list where the first element is a matrix of log-likelihood estimates. Hence we can extract this and calculate the sample variance as follows:

```
## calculate the sample variance
flu_train <- var(flu_train$output)
flu_train
```

```
## [1,] [,1]
## [1,] 59956.58
```

Here the variance is 5.9957×10^4 , which is larger than 3. Hence let's try increasing the number of particles and repeating these steps.

```
## generate numbers of particles to trial
npart <- c(50, 75, 100, 125)

flu_train <- list()
for(i in 1:length(npart)){
  flu_train[[i]] <- PMCMC(
    x = flu_dat,
    priors = priors,
    func = model,
    u = iniStates,
    npart = npart[i],
    iniPars = postMed,
    niter = 500,
    fixpars = TRUE
  )
  flu_train[[i]] <- var(flu_train[[i]]$output)
}
names(flu_train) <- paste0("npart = ", npart)
flu_train <- do.call("c", flu_train)
```

```
flu_train
```

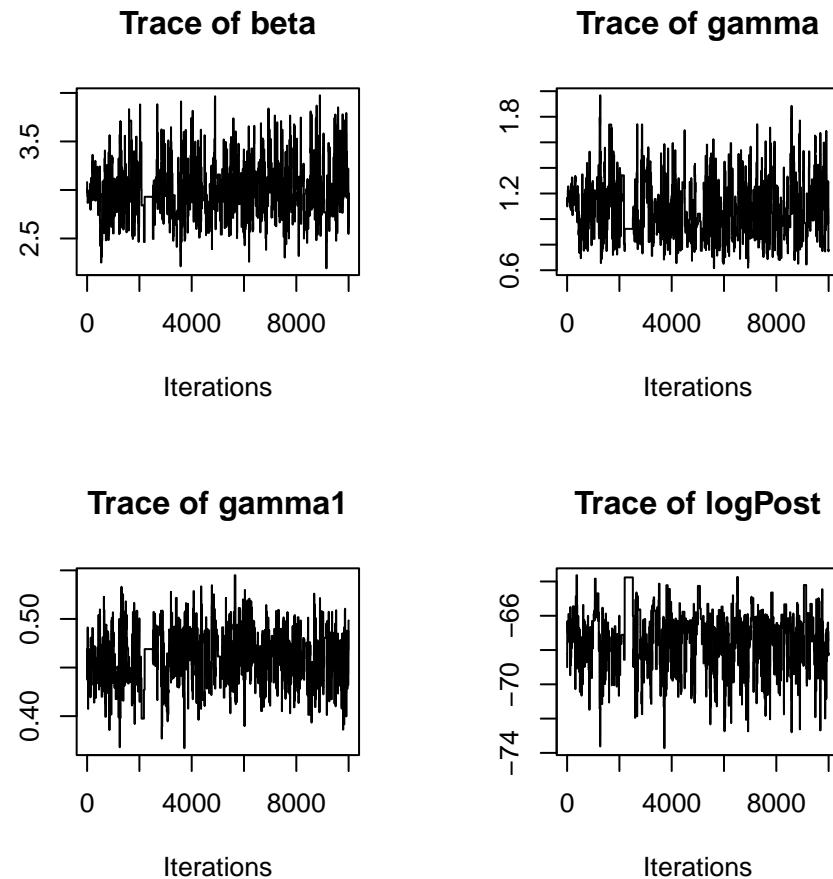
```
##  npart = 50  npart = 75  npart = 100  npart = 125
##    6.559770    3.601458    1.751317    1.409048
```

Here we will choose the number of particles to be 75 (ideally should be a bit larger—but for the sake of exposition we'll tone down a touch). We now start a new chain using 75 particles, and with starting values derived from the training runs.

```
post <- PMCMC(
  x = flu_dat,
  priors = priors,
  func = model,
  npart = 75,
  u = iniStates,
  iniPars = postMed,
  niter = 10000,
  nprintsum = 1000
)
```

(Again I have suppressed the output here for brevity...)

```
## plot and summarise MCMC output
plot(post, "trace")
```

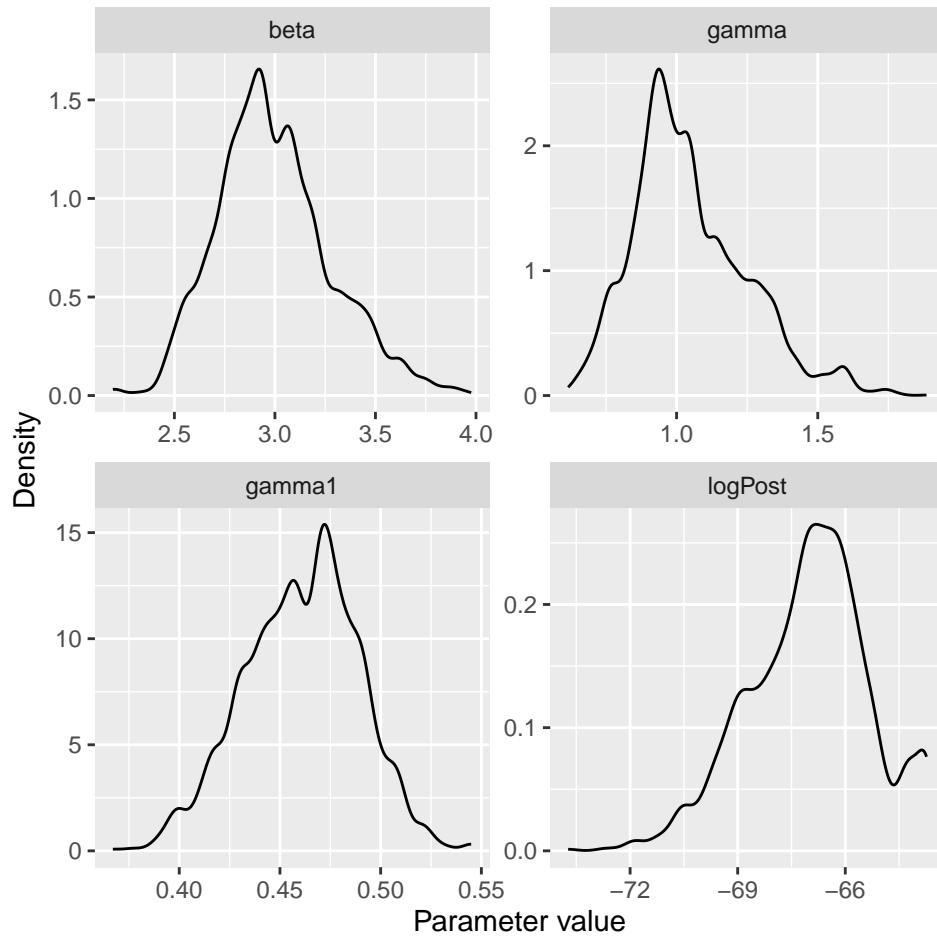
**Note**

We need to run for longer in practice and run multiple chains, but for the sake of time we will proceed with what we have.

10.5 Visualising and summarising the posterior distributions

We can visualise the approximate posterior distributions (after removing some burn-in):

```
## remove burn-in
post <- window(post, start = 2000)
## plot and summarise outputs
plot(post)
```



```
summary(post)
```

```
##  
## Iterations = 1:8001  
## Thinning interval = 1  
## Number of chains = 1  
## Sample size per chain = 8001  
##  
## 1. Empirical mean and standard deviation for each variable,  
##     plus standard error of the mean:  
##  
##           Mean        SD  Naive SE Time-series SE  
## beta      2.9938  0.28385  0.0031733      0.01574  
## gamma    1.0457  0.20216  0.0022601      0.01074  
## gamma1   0.4608  0.02824  0.0003157      0.00157  
## logPost -67.0245 1.68719  0.0188622      0.14081  
##  
## 2. Quantiles for each variable:  
##  
##           2.5%       25%       50%       75%      97.5%  
## beta      2.5032  2.7964  2.9512  3.1595  3.6210  
## gamma    0.7229  0.9132  1.0140  1.1655  1.5381  
## gamma1   0.4014  0.4410  0.4614  0.4805  0.5113  
## logPost -70.5725 -68.0913 -66.8836 -65.9348 -63.7634
```

Task 96

Produce summaries of the posteriors for R_0 and the average length of the infectious period.

Show: Hint on P268

Show: Solution on P259

10.6 Predictive posterior distributions

We can also use the model to predict the future course of an outbreak (with uncertainties). The `SimBIID` package provides a `predict()` method for `PMCMC` objects. To produce predictions we first fit a model to the current available data. This produces a set of posterior samples for each of the parameters. Then, for each set of posterior samples we can produce an estimate of the states of the system at the final observed time point. We do this by running a bootstrap particle filter over the observed time points for each parameter set, and then sampling a trajectory from the weighted set of particles. Hence we also obtain a set of posterior samples for the states of the system at the final observed time point.

Once these have been obtained, we can use the corresponding posterior samples to seed a set of forward simulations into the future up to some pre-determined time point. All of this is done within the `predict()` function; we just need to pass it a suitable `PMCMC` object and a `tspan` argument for the time points we wish to predict to.

As an example, let's pretend that we are at day 3 of the outbreak, and let's fit a model to the observed data up to that time point:

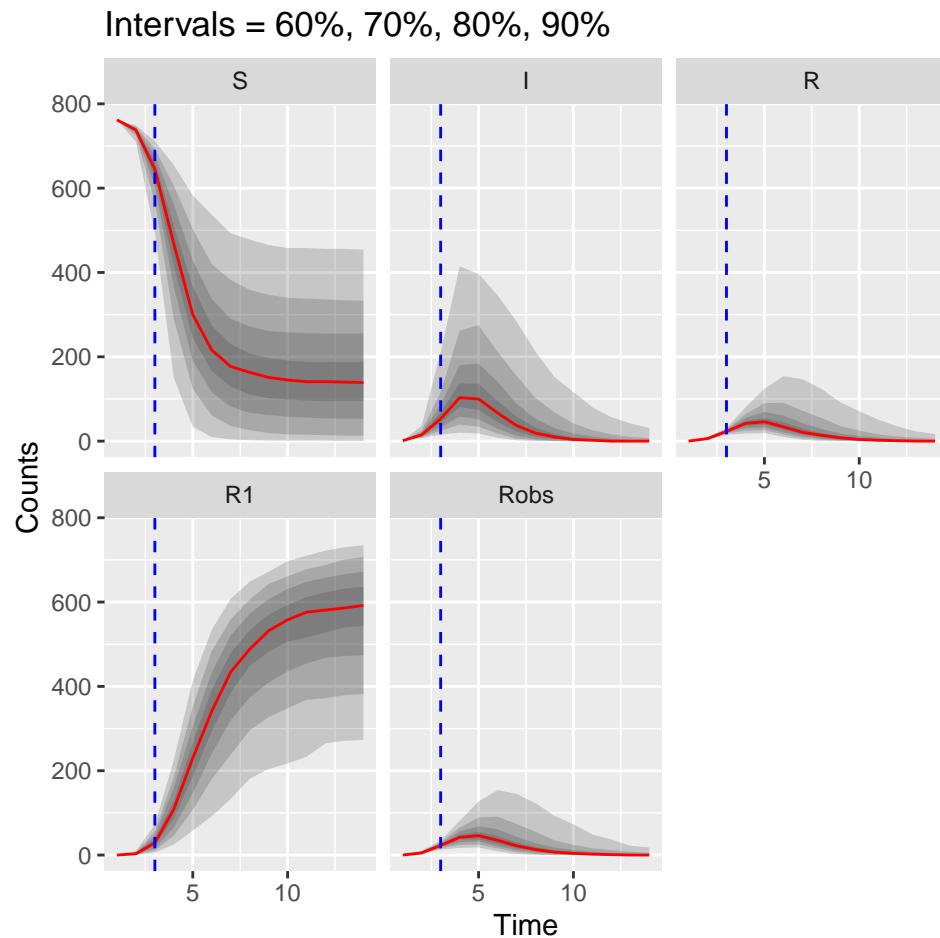
```
## run PMCMC algorithm
post <- PMCMC(
  x = flu_dat[1:3, ],
  priors = priors,
  func = model,
  u = iniStates,
  npart = 75,
  niter = 10000,
  nprintsum = 1000
)
## plot traces
plot(post, "trace")
```

(Output and trace plots suppressed here for brevity...)

Now let's predict forward up to day 14, based on the posterior distributions at day 3. To speed this up we will take 1,000 posterior samples. These can be obtained by using the `window()` function, to remove the first 2,000 iterations as burn-in, and then thin the remaining 8,000 samples by sub-sampling every 8th sample. The `predict()` function produces a `SimBIID_runs` object, which we can plot as before. Since `obsProcess` was specified in the model, the `predict()` function will also produce predictions that take the `observation` process into account. Here the observation process acts only on the `R` class, and so this will produce an extra column called `Iobs` here, which contains predictions assuming a Poisson observation error around the simulated `R` counts (called `Rob`s here which is specified in the `datNames` column of the original `obsProcess` object).

```
## run predictions forward in time
post_pred <- predict(window(post, start = 2000, thin = 8), tspan = 4:14)

## plot predictions
plot(post_pred, quant = c(0.6, 0.7, 0.8, 0.9))
```



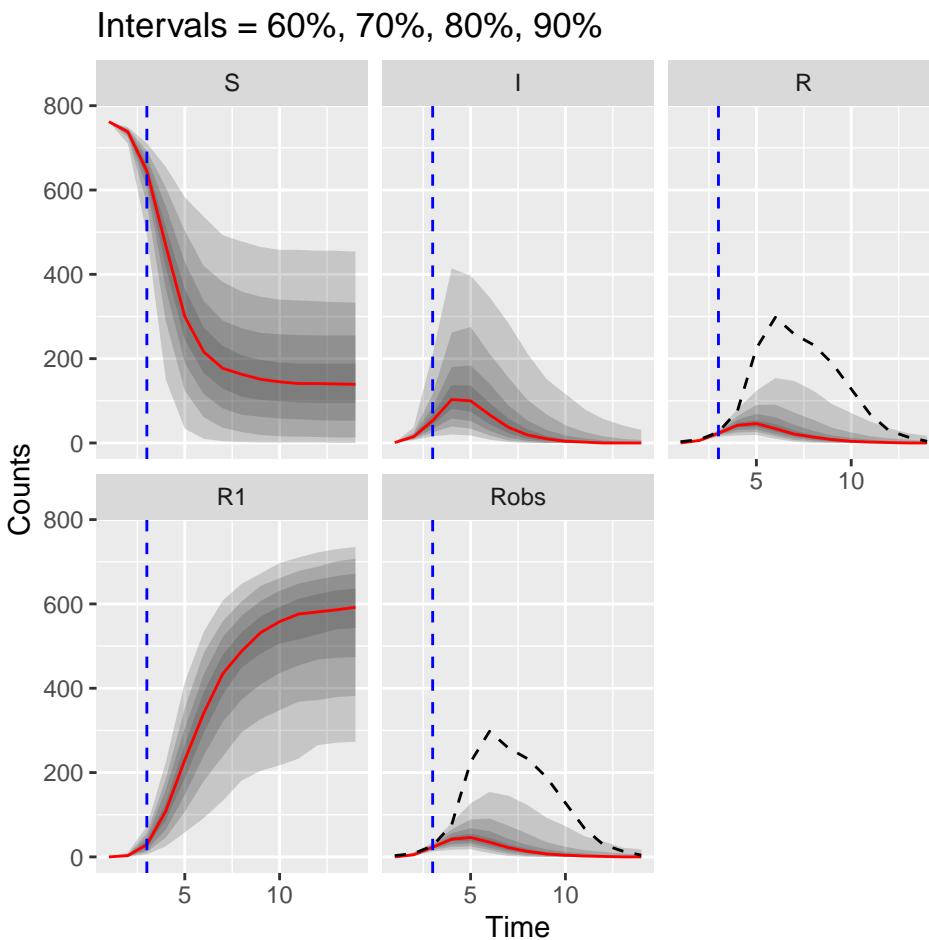
The uncertainties up to the blue dashed line are derived from the bootstrap particle filter, whereas the uncertainties going forward are from direct simulations from the model. Since the *R* curve can be compared directly to the observed data, we can add the observed data in as additional arguments to the `plot()` method here. We just have to add an additional `matchData` argument to tell the function which columns of the data to plot against which output from the model. In this case we pass the complete data to the function, just so that we can see how close the predictions (estimated from the model fitted at the dashed blue time point) were to the actual data. If you were doing this in real time you would only have the data up to the dashed blue time point.

Note

The `matchData = c("Robs = R")` below tells the `plot()` function to match the column called `Rob`s in the data set to the `R` class from the simulations. It might be worth plotting the observations against the `Rob`s output from the simulations also, since the simulated `Rob`s curves include the **observation process**.

```
## plot predictions and add observed I curve
plot(post_pred, quant = c(0.6, 0.7, 0.8, 0.9),
     data = flu_dat, matchData = c("Robs = R", "Robs = Robs"))
```

Warning: `cols` is now required when using `unnest()`.
 ## i Please use `cols = c(value)`.



Task 97

Repeat the above procedure, refitting at days 5, 8 and 11. What happens to the uncertainties in the predictions? Why is this so?

Show: Solution on P259

Part VI

Additional Materials


```
## [conflicted] Will prefer SimInf::run over any other package.
```


Chapter 11

Estimating R_0

Andrew Conlan (ajkc2@cam.ac.uk)

Aim: To compare final size and exponential growth estimators for R_0 using simulated data.

Outline:

1. Introduction: models and data sets
2. Final size method
3. Regression method
4. RECON `earlyR` method

Important

All required data and script files can be downloaded [here](#). Please download this ZIP file and extract the files to your working directory.

11.1 Introduction: models and data sets

In Epidemic Practicals 1, 2 & 3 you wrote functions to numerically solve and simulate the deterministic and stochastic SIR models. For this practical we have extended these functions to add a latent class and implement the SEIR epidemic model introduced in lectures. If you have time at the end of this practical you can use these functions to simulate your own “Outfluenza” and “Biggles” data and explore the performance of the final size and linear regression methods.

However, so we all get the same results, we have provided simulated epidemics of “Outfluenza” and “Biggles” from which we will estimate R_0 using the final size, (log)-linear regression and the `earlyR` method from the **R Epidemics Consortium** (RECON, <https://www.repidemicsconsortium.org/>).

The simulated outbreaks are provided as **line lists**, a common instrument used to collect data on individual cases during an outbreak where the date of infection (or rather the notification of a case) is recorded along with other relevant epidemiological information. For our purposes, the date of notification is sufficient to reconstruct the epidemic curve and test our different estimators of R_0 .

Begin by reading in the line lists and converting the dates (stored as character strings) to date objects:

```

require(chron)
require(incidence)
# Load line lists of cases for simulated outbreaks
outfluenza1 <- read.table('Outfluenza1_cases.dat')
outfluenza1$x = as.Date(outfluenza1$x)

outfluenza2 <- read.table('Outfluenza2_cases.dat')
outfluenza2$x = as.Date(outfluenza2$x)

biggles <- read.table('biggles_cases.dat')
biggles$x = as.Date(biggles$x)

```

We are going to use the `incidence` package from RECON which has been written to simplify computing, visualising and modelling incidence of infectious disease from dated event data such as line lists.

To illustrate the key functions and concepts we will use `incidence` to construct daily incidence and cumulative incidence curves for the first outfluenza outbreak.

The main workhorse function is called `incidence` which converts a list of dates into an `incidence` object that bins cases into a given interval:

```
outfluenza1.i = incidence(outfluenza1$x, interval=1)
```

We can see a summary of the incidence object by simply typing the name of the variable:

```
outfluenza1.i
```

```

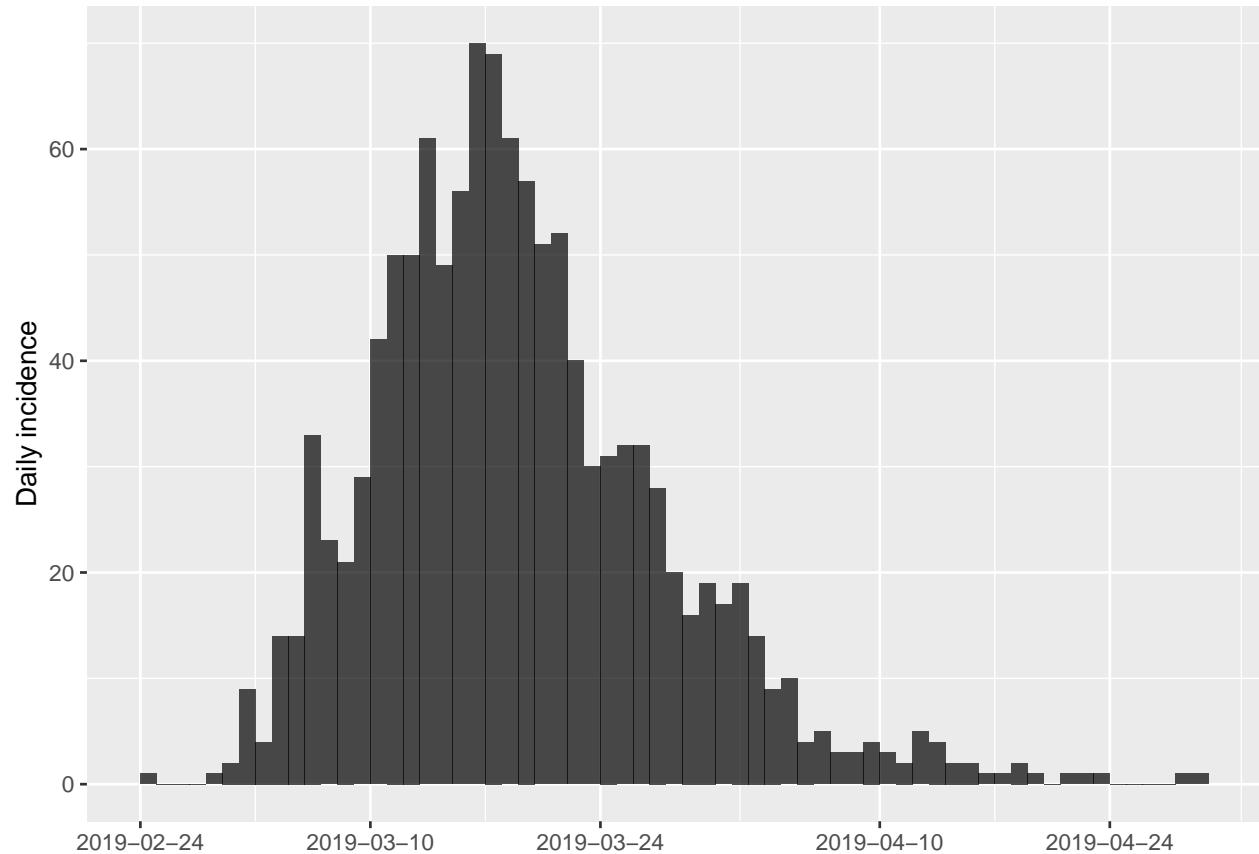
## <incidence object>
## [1] 1183 cases from days 2019-02-24 to 2019-04-29
##
## $counts: matrix with 65 rows and 1 columns
## $n: 1183 cases in total
## $dates: 65 dates marking the left-side of bins
## $interval: 1 day
## $timespan: 65 days
## $cumulative: FALSE

```

So, the outfluenza1 outbreak consists of 1183 cases over the course of 65 days. We have chosen to bin cases on a daily interval (which is also the default so we could have simply left out this argument and will do so from now on).

The `incidence` package provides custom plot functions for `incidence` objects:

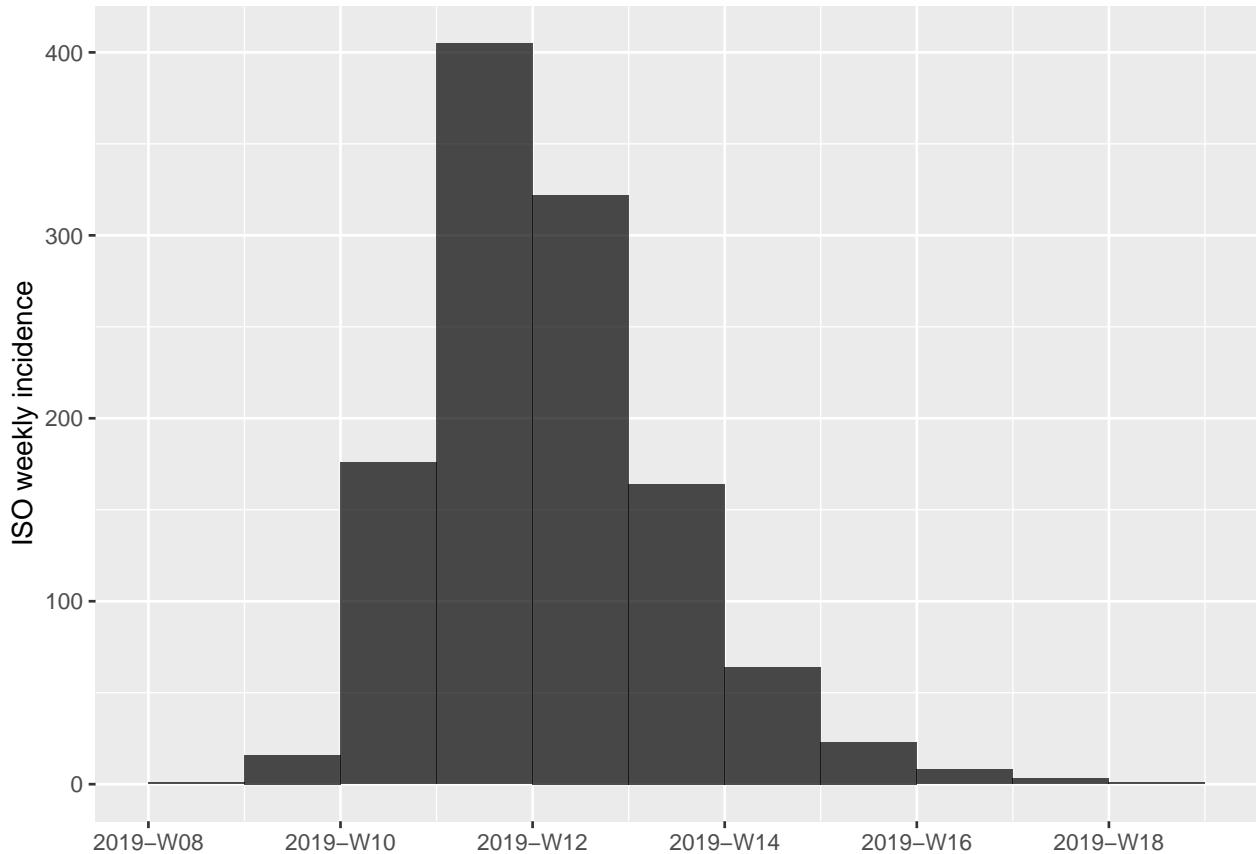
```
plot(outfluenza1.i)
```



Which should be familiar from the lectures.

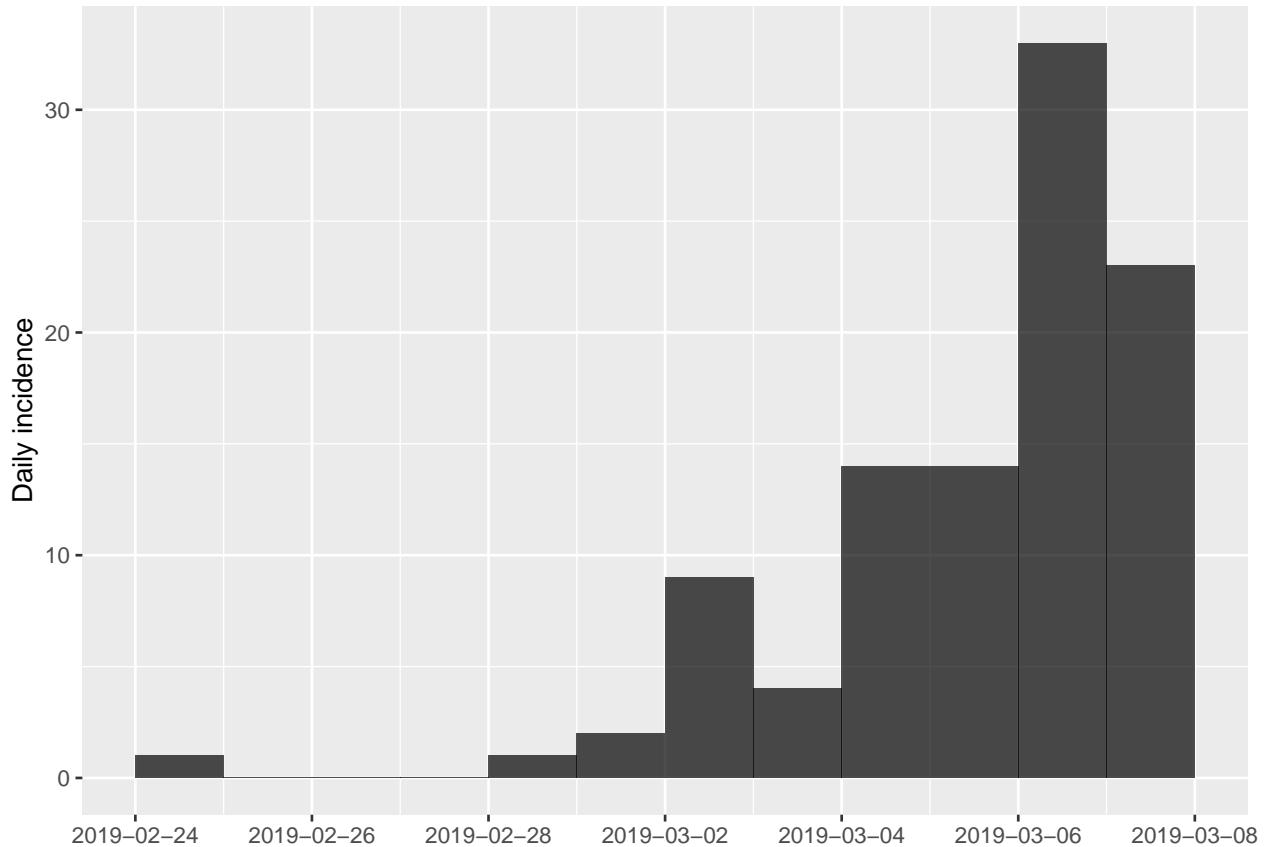
You can try experimenting with different intervals to see how this affects the shape of the epidemic curve:

```
plot(incidence(outfluenza1$x, interval=7))
```



`incidence` also allows you to subset curves between start and end dates:

```
plot(subset(outfluenza1.i, from=outfluenza1$x[1], to=outfluenza1$x[100]))
```

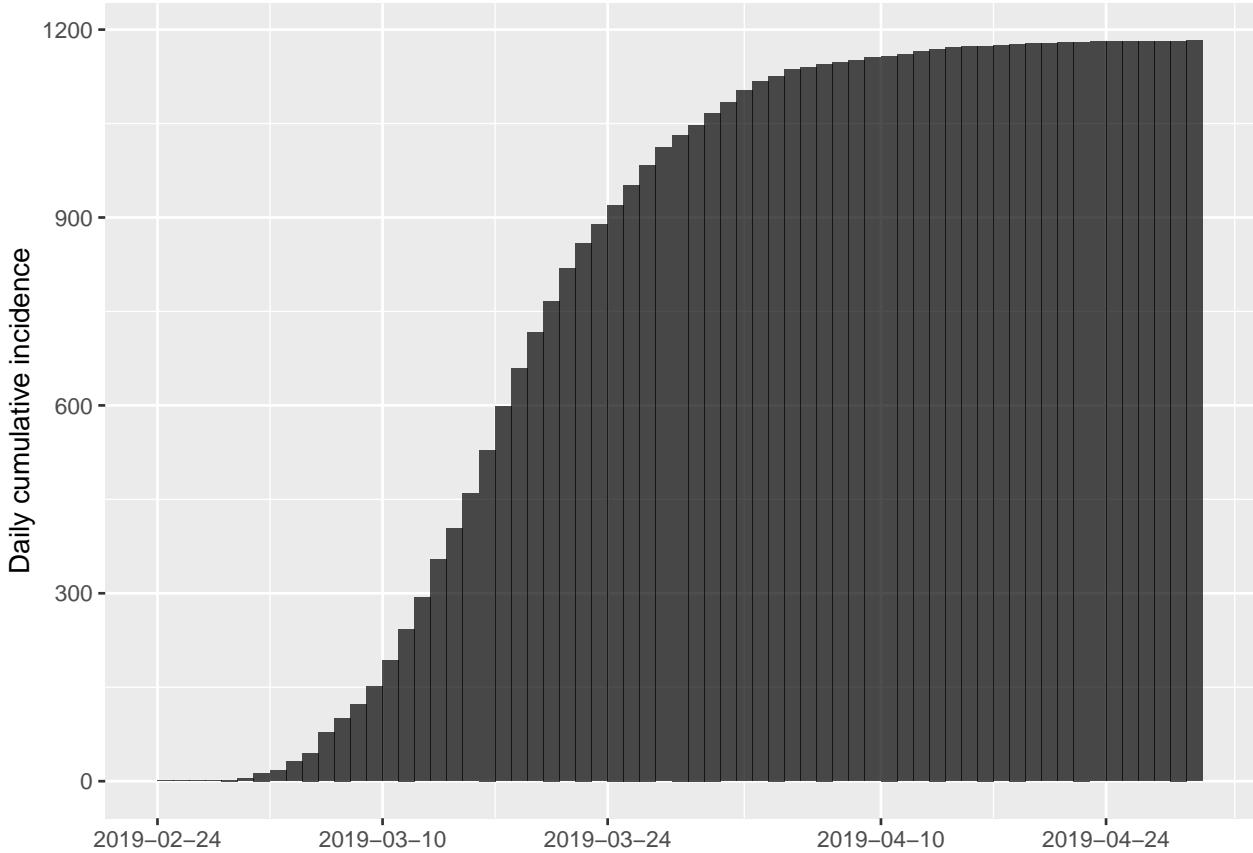
**Task 98**

What is the significance of choosing the dates to subset in this way?

Show: Solution on P262

The `cumulate()` function returns a the cumulative incidence curve:

```
plot(cumulate(outfluenza1.i))
```



Finally, you can convert an `incidence` object to a data frame by using the `cast` function `data.frame()`:

```
outfluenza1.df = data.frame(outfluenza1.i)
head(outfluenza1.df)
```

```
##          dates counts
## 1 2019-02-24      1
## 2 2019-02-25      0
## 3 2019-02-26      0
## 4 2019-02-27      0
## 5 2019-02-28      1
## 6 2019-03-01      2
```

11.2 Final size method

Final size methods for R_0 are in themselves a whole field of research. In this practical we will consider the utility of the basic deterministic “first estimate” also known as the final size formula:

$$R_0 = -\frac{\ln(1 - z_f)}{z_f} \quad (11.1)$$

where z_f is the total fraction of the population infected at the end of an epidemic in a closed population.

Task 99

Write a function `R0final` that returns the value of R_0 for a given value of z_f . Is there any restriction on the values that z_f can take for the final size equation to be valid? Your function should check that z_f takes a valid value and return a missing value (NA) when the final size equation is not defined.

Show: Solution on P263

This formula is more commonly used to predict the final size (z_f) of an epidemic once we have an independent estimate of R_0 . This is less straightforward as we cannot solve equation (11.1) explicitly for z_f . We can progress by rewriting (11.1) into a problem we can solve numerically:

$$R_0 + \frac{\ln(1 - z_f)}{z_f} = 0 \quad (11.2)$$

We can find the final size (z_f) corresponding to a given value of R_0 , by solving for the value(s) of z_f for which (11.2) is equal to zero. Finding the roots of a function is a common problem, so it should not surprise you that there is a R function to do the job. `uniroot` uses an numerical procedure to estimate the roots of an arbitrary function `f` within a fixed `interval=c(lower,upper)`. We can use `uniroot` to write a function that returns the predicted final size for a given value of R_0 :

```
FinalSize <- function(R0) {
  return(uniroot(function(zf)
    {R0+log(1-zf)/zf},
    interval=c(.Machine$double.xmin, 1.0))$root)
}
```

Check this function works by trying these test values on the R terminal:

```
FinalSize(1.2)
```

```
## [1] 0.3136976
```

```
FinalSize(1.0)
```

```
## [1] 6.103516e-05
```

```
FinalSize(0.9)
```

```
## [1] 6.103516e-05
```

The exact values may differ slightly between versions of R on different machines.

Note

Numerical methods are only accurate up to a specified precision or **tolerance**. This is fundamentally limited by the precision with which real numbers can be represented on a computer (that naturally works with integer or binary numbers). `.Machine$double.xmin` is the smallest number (or difference between two numbers) that can be stored in a given version of R. This value may change between versions and on different computer platforms (Windows, Mac OS, Linux...)

Task 100

Use the auxiliary functions of the incidence package and these new functions you have just written to estimate R_0 for the exemplar outfluenza and biggles outbreaks. (Remember the school size for all of these outbreaks was 1300 children.)

Show: Solution on P263

11.3 Regression method

In the lectures we discussed how the early phase of an epidemic can be approximately modelled by an exponential growth model:

$$I(t) = I(0)e^{\Lambda t} \quad (11.3)$$

where the exponential rate can be related to R_0 with the functional form depending on the structure of the epidemic model (in particular with respect to the distribution of latent and infectious periods). If we take logs of both sides of equation (11.3) and rearrange we get the equation of a straight line:

$$\log(I(t)) = \Lambda t + \log(I(0)) \quad (11.4)$$

with y-intercept given by the constant $\log(I(0))$ and slope Λ . So, we can therefore obtain a first approximation to R_0 simply by estimating the slope of the (logged) epidemic curve.

In principle, estimating R_0 using this method could be as straightforward as plotting the incidence or cumulative incidence curve on logarithmic graph paper and fitting a “best-fit” line, or using the Solver in Excel. We can be a little more sophisticated and use the linear regression model (`lm`) function in R. `lm()` uses a least squares method, effectively optimising the fit of the straight line to minimise the squared error between the line and the data.

As the exponential approximation is only valid early in the epidemic, including data from the full epidemic curve would bias our estimate of R_0 . As discussed in the lecture we will use the first 100 cases. For a given simulation we need to select all the data-points up to the 100th case. We can achieve this by sub setting the incidence object as before:

```
outflu1.sub<- data.frame(subset(outfluenza1.i, from=outfluenza1$x[1], to=outfluenza1$x[100]))
```

We fit a linear model (best straight line fit) to find the slope of the log cumulative cases as described in the lectures:

```
# Fit straight line to plot of biggles$time (time) and log of the cumulative cases (C)
outflu1.fit = lm(log(1+counts) ~ dates, data=outflu1.sub)
```

Note:

We add 1 to counts to handle any zero cases (this should not affect the estimated slope).

We can see the result of the regression by using the `summary` function:

```
summary(outflui.fit)

##
## Call:
## lm(formula = log(1 + counts) ~ dates, data = outflui.sub)
##
## Residuals:
##     Min      1Q  Median      3Q     Max 
## -0.7081 -0.3553 -0.1211  0.3675  0.9870 
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -5.996e+03  7.855e+02 -7.633 1.77e-05 ***
## dates        3.340e-01  4.374e-02   7.635 1.77e-05 ***  
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5231 on 10 degrees of freedom
## Multiple R-squared:  0.8536, Adjusted R-squared:  0.8389 
## F-statistic:  58.3 on 1 and 10 DF,  p-value: 1.768e-05
```

Task 101

The `summary()` function provides a lot of detailed information on the statistical fit of the regression line – for the purpose of estimating R_0 the key value is the estimated slope highlighted in bold above. Using the expression for the SIR model presented in the lectures calculate the R_0 of outfluenza using this slope:

Show: Solution on P263

We can estimate the uncertainty in our estimate of R_0 from the uncertainty in our estimate of the slope from the linear regression. The R function `confint()` will calculate 95% confidence intervals for our regression model:

```
confint(outflui.fit)
```

```
##              2.5 %      97.5 %
## (Intercept) -7745.8850733 -4245.5519457
## dates        0.2365213    0.4314554
```

Task 102

Use these confidence intervals to calculate a 95% confidence interval for your estimate of R_0 .

Show: Solution on P263

The incidence package has built in functions that simplify estimating the exponential growth rate. The base function is `fit` returns an `incidence_fit` object which returns details of the fit including the estimated exponential growth rate r and the confidence interval:

```
outflu1.fit2 <- fit(outfluenza1.i)
outflu1.fit2
```

```
## <incidence_fit object>
##
## $model: regression of log-incidence over time
##
## $info: list containing the following items:
##   $r (daily growth rate):
## [1] -0.0504102
##
##   $r.conf (confidence interval):
##       2.5 %    97.5 %
## [1,] -0.06954703 -0.03127338
##
##   $halving (halving time in days):
## [1] 13.75014
##
##   $halving.conf (confidence interval):
##       2.5 %    97.5 %
## [1,] 9.966597 22.16413
##
##   $pred: data.frame of incidence predictions (57 rows, 5 columns)
```

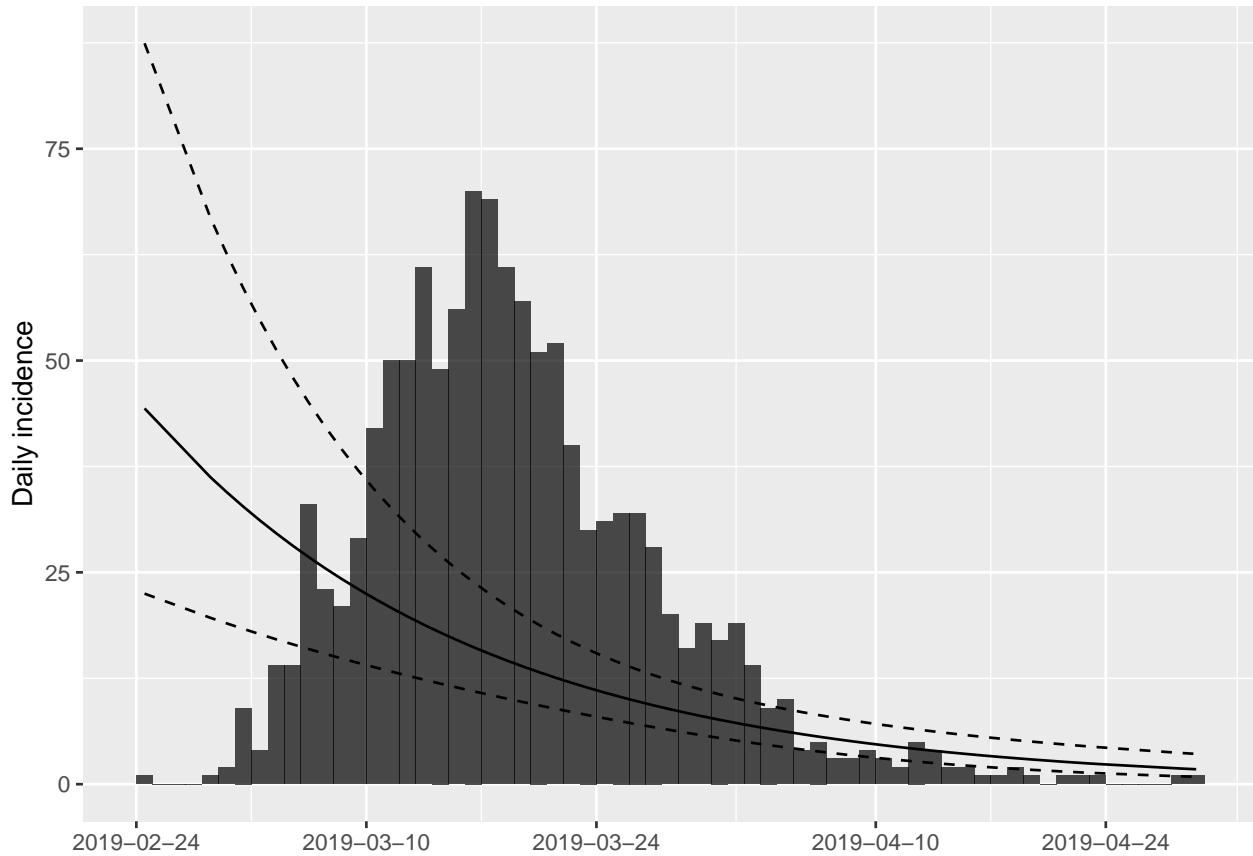
Question 103

Why does `fit` ignore dates with 0 incidence?

Show: Answer on P263

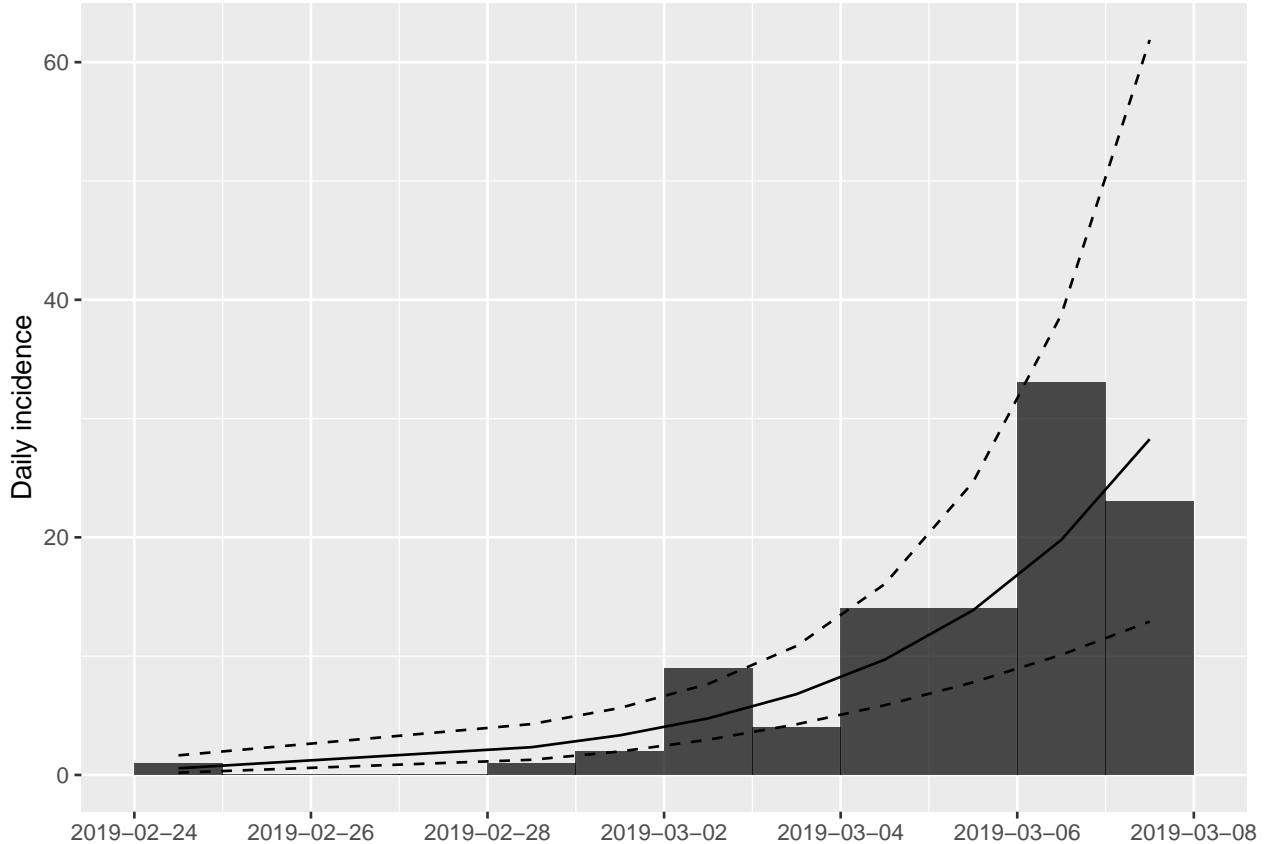
The estimated r is much lower than we obtained by our manual estimate. To see what has happened we can add the obtained fit to a plot of the incidence:

```
plot(outfluenza1.i, fit= outflu1.fit2)
```



We can use the `subset` function to limit our fit to the first 100 cases as before:

```
outflu1.sub = subset(outfluenza1.i, from=outfluenza1$x[1], to=outfluenza1$x[100])  
outflu1.fit3 = fit(outflu1.sub)  
plot(outflu1.sub, fit=outflu1.fit3)
```



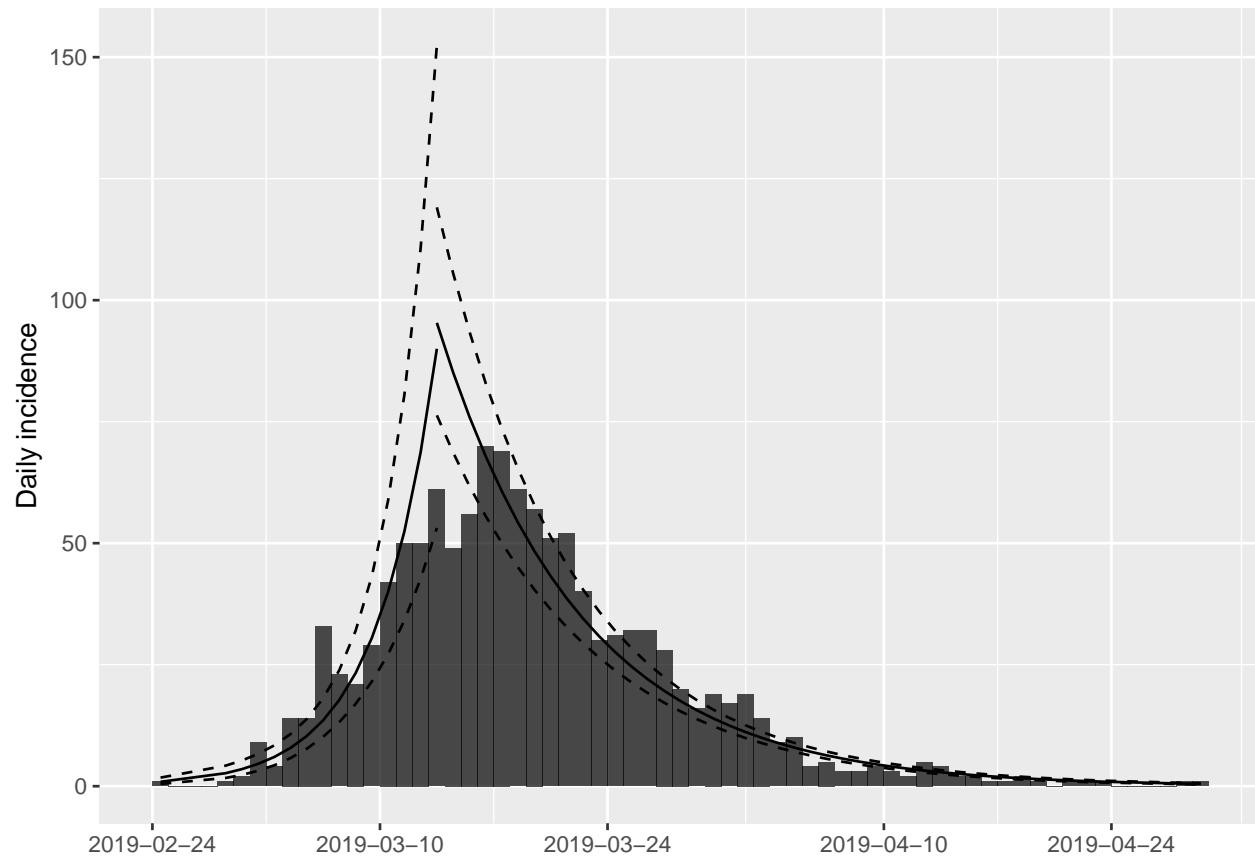
Question 104

How do the estimates from the incidence package and our manual estimate compare to each other and why (what's different)?

Show: Answer on P264

The `incidence` package provides a method for automatically sub setting the data by finding the optimum fit of two log-linear models (minimising the squared error of both):

```
outflui1.fit5 = fit_optim_split(outfluenza1.i)
plot(outfluenza1.i,fit=outflui1.fit5$fit)
```



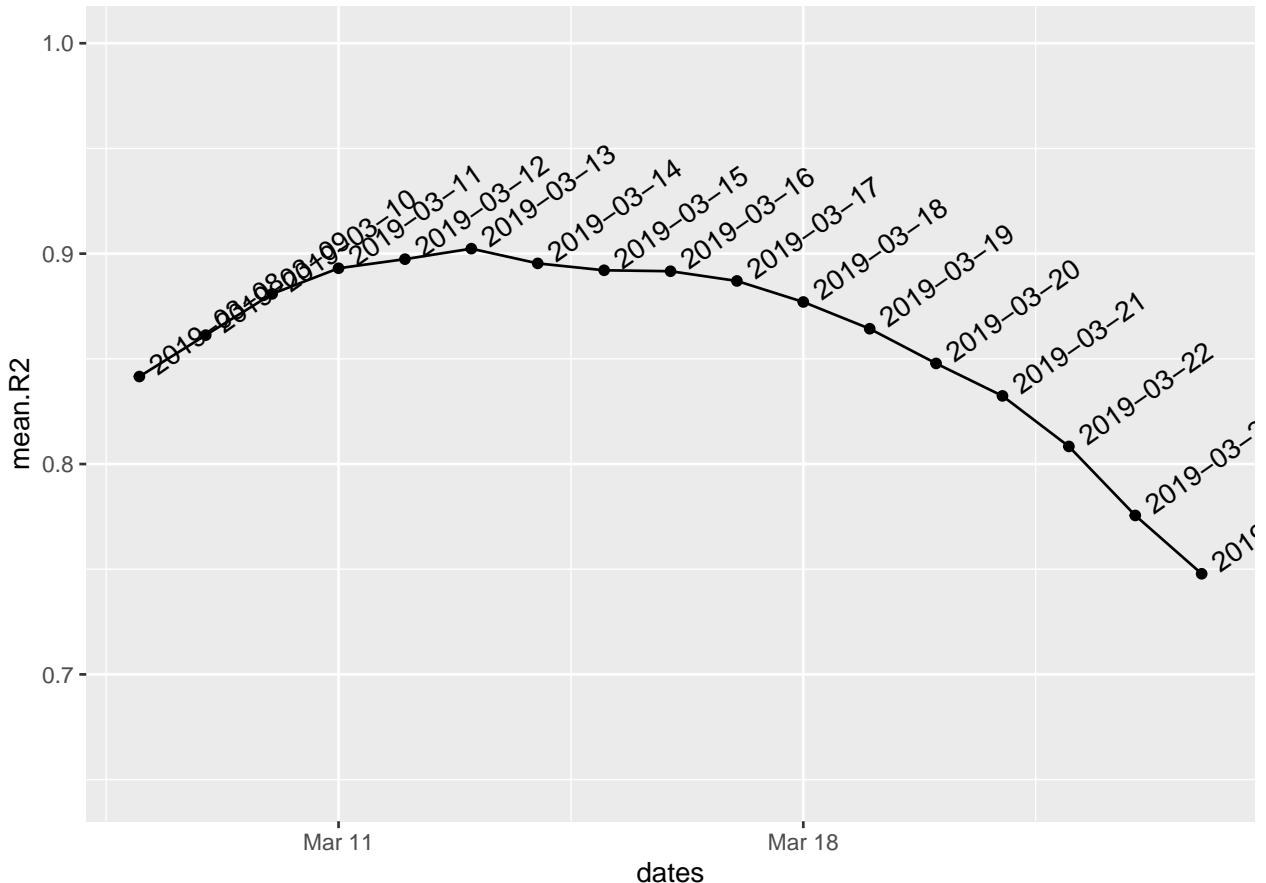
Note

The output of `fit_optim_split` now continues information on the two log-linear model fits, the first for the attack of the epidemic, the second describing the decay after the epidemic peak (which is also estimated as the **changepoint** between the two models).

Examine the output by entering into the R console:

```
outflu1.fit5
```

Which should also generate a plot of the mean r-squared error (R^2) of the two log-linear models against different dates for the changepoint:



11.4 RECON earlyR method

Finally we will use the `earlyR` package implementation of the Teunis and Wallinga method to estimate R_0 and compare to the other methods. The key function of the `earlyR` package is the `get_R` function that returns an estimate of R_0 when passed an incidence object and a serial interval distribution. The serial interval can be specified manually, but we will use the default gamma distribution which is parameterised by the mean (`si_mean`) and standard deviation (`si_sd`). For outfluenza, the serial interval distribution is exponentially distributed with mean equal to the variance (`si_mean = 5, si_sd = 5`).

```
library(earlyR)
estR = get_R(outfluenza1.i, si_mean=5, si_sd=5)
estR

##
## /// Early estimate of reproduction number (R) //
## // class: earlyR, list
##
## // Maximum-Likelihood estimate of R ($R_ml):
## [1] 1.001001
##
##
## // $lambda:
##    NA 0.1812692 0.1484107 0.1215084 0.09948267 0.2627188...
```

```

## 
## // $dates:
## [1] "2019-02-24" "2019-02-25" "2019-02-26" "2019-02-27" "2019-02-28"
## [6] "2019-03-01"
## ...
##
## // $si (serial interval):
## A discrete distribution
##   name: gamma
##   parameters:
##     shape: 1
##     scale: 5

```

Once again we have obtained a much lower estimate of R ($R_0 = 1.0$) than expected as the `earlyR` method is only valid during the exponential phase of the epidemic. If we subset again using the first 100 cases you should obtain a more consistent estimate of R_0 of 2.43.

We can calculate a bootstrapped 95% confidence interval for the estimate using the `sample_R` function:

```
quantile(sample_R(estR), c(0.025, 0.975))
```

```

##      2.5%    97.5%
## 0.9557057 1.0663163

```

So our `earlyR` estimate of $R_0 = 2.43$ (2.0–3.0, 95% CI).

Task 105

Now we have introduced all of the methods, let's compare the estimates (and confidence intervals where possible) of R_0 for our three exemplar outbreaks. Complete the following table by adapting the code you have used in the practical.

Remember that for Outfluenza the mean serial interval is 5 days (standard deviation 5 days). For Biggles the latent period is 5 days, the exposed period is 5 days. As discussed in lectures, adding two exponentially distributed variables together gives a gamma distributed variable. So the serial interval for Biggles will be gamma distributed with a scale parameter of 10 days and a shape parameter of 2, giving a mean serial interval of 10 and standard deviation of $5\sqrt{2}$. (Ask your instructor for more details if interested!)

Outbreak	Regression r	Regression R_0	Final Size R_0	<code>earlyR</code> R_0
Outfluenza1				
Outfluenza2				
Biggles				

Task 106

Compare your estimates from the three methods of estimation. Are you surprised by the variability in estimates of R_0 that you have obtained compared to the size of the 95% confidence intervals of the estimates?

Show: Solution on P264

11.5 Appendix

You might be interested now in explore how well the final size and regression estimators perform on different stochastic replicates of the SIR or SEIR models.

`SIRmodels.R` and `SEIRmodels.R` files provide functions to allow you to generate your own simulated epidemics of Biggles and Outfluenza.

To load the functions into the R workspace using the following R code:

```
source('SIRmodels.R')
source('SEIRmodels.R')

N = 1300
```

`SIRmodels.R` provides two familiar functions:

- `DetSIR.dyn`

Uses the internal R function `lsoda()` to numerically solve the **deterministic** SIR model given 5 parameters:

- N: Population size
- I₀: Initial infected
- B: transmission rate
- M: “recovery” rate = $1/T_I$ corresponding to average infectious period of T_I
- f_time: run time for numerical solution

- `create.StocSIR` defines an stochastic SIR model to use with the `SimInf` package and takes 6 arguments:

- N: Population size
- I₀: Initial infected
- beta: transmission rate
- m: “recovery” rate = $1/T_I$ corresponding to average infectious period of T_I
- f_time: run time for numerical solution
- reps: number of replicates to simulate

`SEIRmodels.R` provides equivalent functions `DetSEIR.dyn` and `StocSEIR.dyn` that implement the deterministic and stochastic SEIR epidemic model respectively.

Both of these functions implementing the SEIR model take 7 parameters in the order that follows:

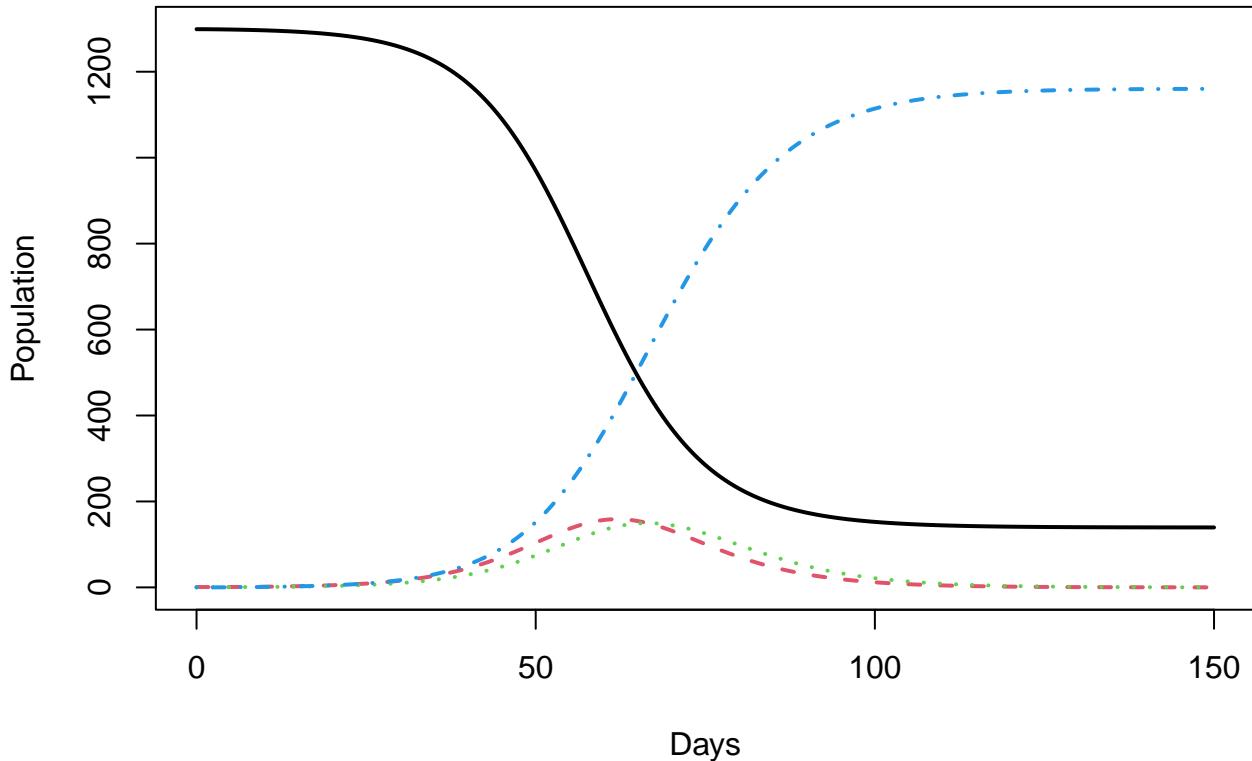
- N: Population size
- I₀: Initial infected
- E₀: Initial exposed
- B: transmission rate
- G: “progression” rate = $1/T_E$, with average exposed period of T_E
- M: “recovery” rate = $1/T_I$ corresponding to average infectious period of T_I
- f_time: run time for numerical solution

The deterministic equations for the SEIR are then:

$$\begin{aligned}
 \frac{dS}{dt} &= -\frac{\beta SI}{N} \\
 \frac{dE}{dt} &= \frac{\beta SI}{N} - gE \\
 \frac{dI}{dt} &= gE - mI \\
 \frac{dR}{dt} &= mI
 \end{aligned}$$

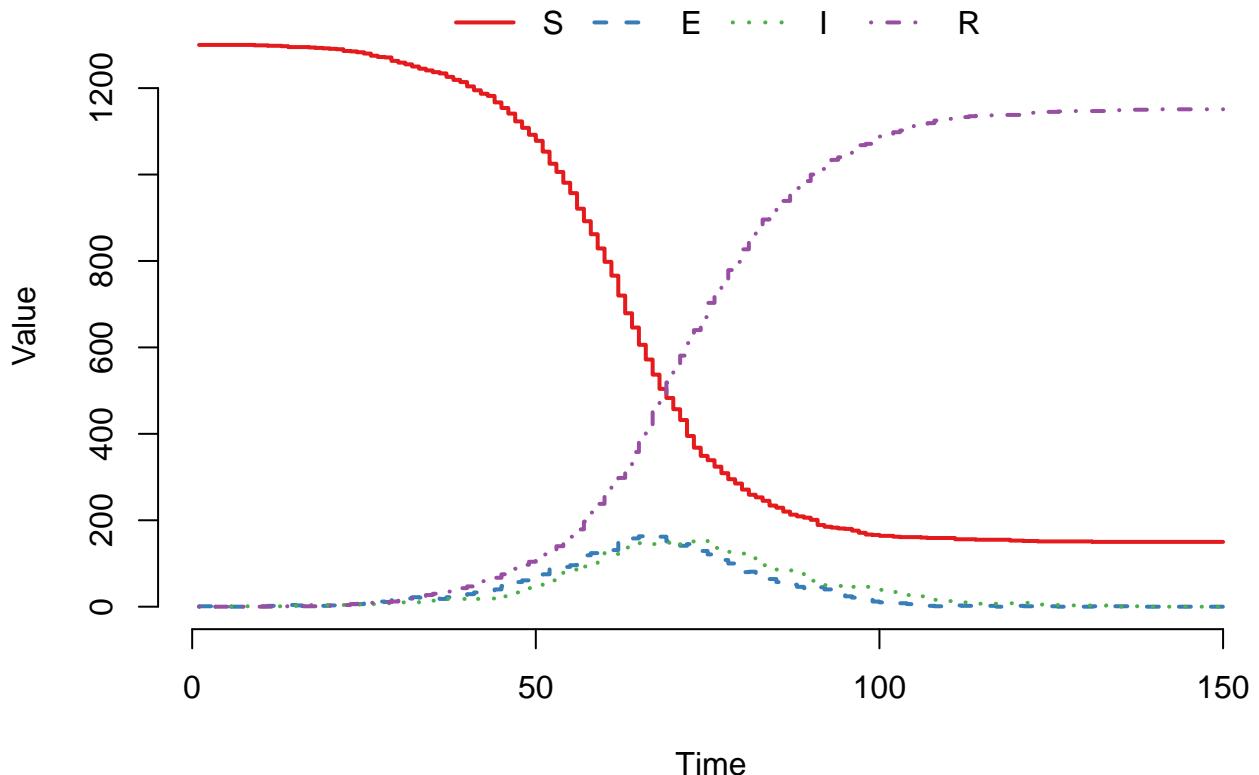
You can check this new code for the deterministic *SEIR* model by simulating an epidemic using the Biggles parameters from the lecture and comparing to the figures in the lecture notes:

```
# Biggles Exemplar Epidemic
det.sol<-DetSEIR.dyn(1300, 1, 0, 0.5, 1/5.0, 1.0/5.0, 150)
matplot(det.sol[,1],det.sol[,2:5],type='l',xlab='Days',ylab='Population',lwd=2)
```



Checking the stochastic model is more difficult as the output will be different every time we run it! As a first sanity check we can compare stochastic simulations to the deterministic epidemic, checking that they (roughly) scatter evenly around the “mean” behaviour:

```
SEIRmodel <- create.StocSEIR(1300, 1, 0, 0.5, 1.0/5.0, 1/5.0, 150, 1)
out <- run(model=SEIRmodel)
plot(out)
```



To simplify working with the incidence package we also provide a wrapper function `line_list()` that takes a `SimInf` object (`out`) as argument, along with a numeric value (`node`) and returns a line list of dates of cases for specified node:

```
SEIRmodel <- create.StocSEIR(1300, 1, 0, 0.5, 1.0/5.0, 1/5.0, 150,100)
out <- run(model=SEIRmodel)
line_list(out,1)
```

Chapter 12

Seasonality and Measles Epidemics

Andrew Conlan (ajkc2@cam.ac.uk)

12.1 Measles data and challenge

Implement a deterministic model to predict the impact of vaccination on the timing of outbreaks of measles in London. You should make the following assumptions to proceed:

- Constant population size of 3.3 million
- Birth rate of 20 per thousand per year
- Basic reproduction number $R_0 = 17$ (for sinusoidal forcing model)
- Cases can be approximately calculated from the number of infectives by multiplying by 7/5 (i.e. reporting period /average infectious period)
- The reporting rate during this period was 40%

A time-series of measles cases from London from 1950–1964 (immediately prior to the introduction of vaccination in the UK) are included as part of the `tsiR` package in R that provides historical time-series data from England and Wales along with functions to work with the so-called time-series *SIR* model (*TSIR*)—a discrete time chain binomial model that can be used to very successfully model and predict measles dynamics (and to a lesser extent other strongly immunizing childhood infections).

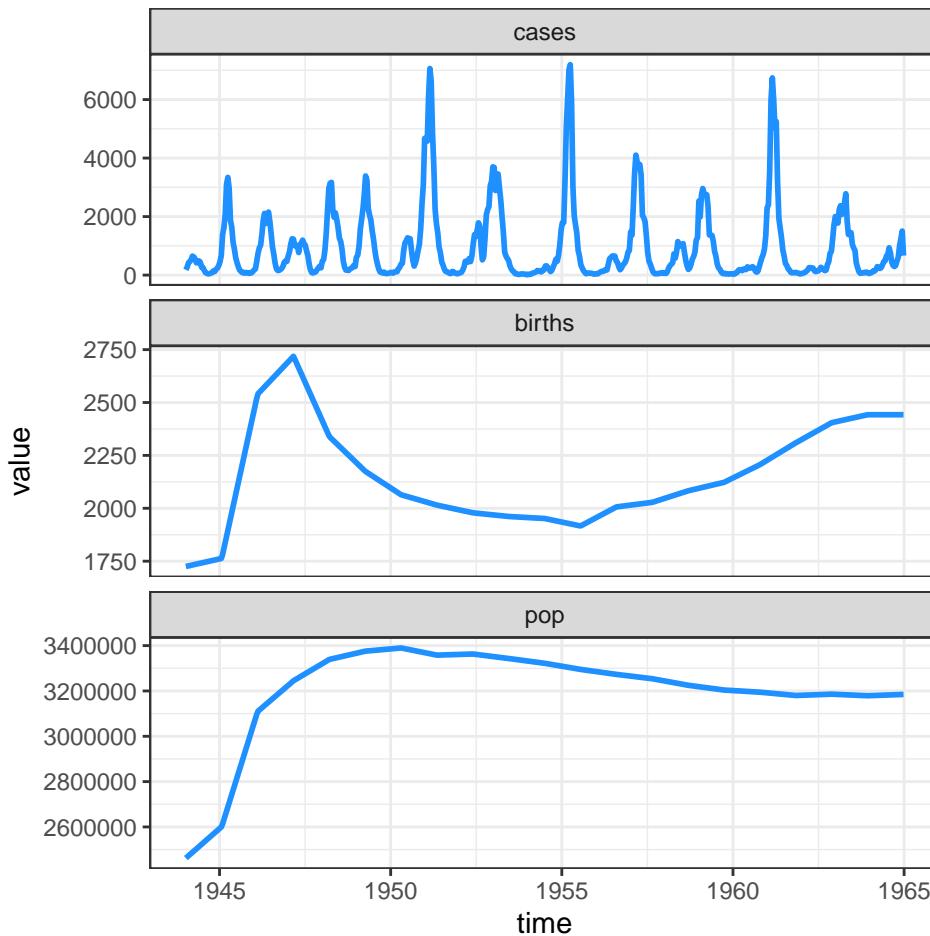
The `plotdata` function from the `tsiR` package provides a summary of the incidence and demographic data (birth rates and population size):

```
require(tsiR)
require(tidyverse)
require(ggplot2)
require(patchwork)
require(deSolve)

data("twentymeas")

LondonMeas <- twentymeas[["London"]]

plotdata(LondonMeas)
```



As you can see, our simplifying assumption of fixed birth and population size glosses over the “baby boom” post **WWII** that pushed the typical two-year cycle of measles epidemics into annual outbreaks and led (along with migration) to an increase in the population size. The *TSIR* model can be used to estimate the seasonal variation in transmission rates presented in the background slides (on which we overlay the typical pattern of school terms in England and Wales calculated by the `mk_terms` function below):

```
LondonRes <- runtsir(data=LondonMeas, IP = 2,
                      xreg = 'cumcases', regtype='gaussian',
                      alpha = NULL, sbar = NULL,
                      family = 'gaussian', link = 'identity',
                      method = 'negbin', nsim = 100)
```

```
##           alpha      mean beta      mean rho      mean sus
## 9.60e-01    1.19e-05   4.57e-01  1.14e+05
## prop. init. sus. prop. init. inf.
## 3.01e-02    6.12e-05
```

```
mk_terms <- function(beta, alpha)
{
  s = 273.0/364.0
  bh = (beta*(1 + 2*alpha*(1-s)))
  bl = (beta*(1 - 2*alpha*s))
```

```

terms = numeric(364)

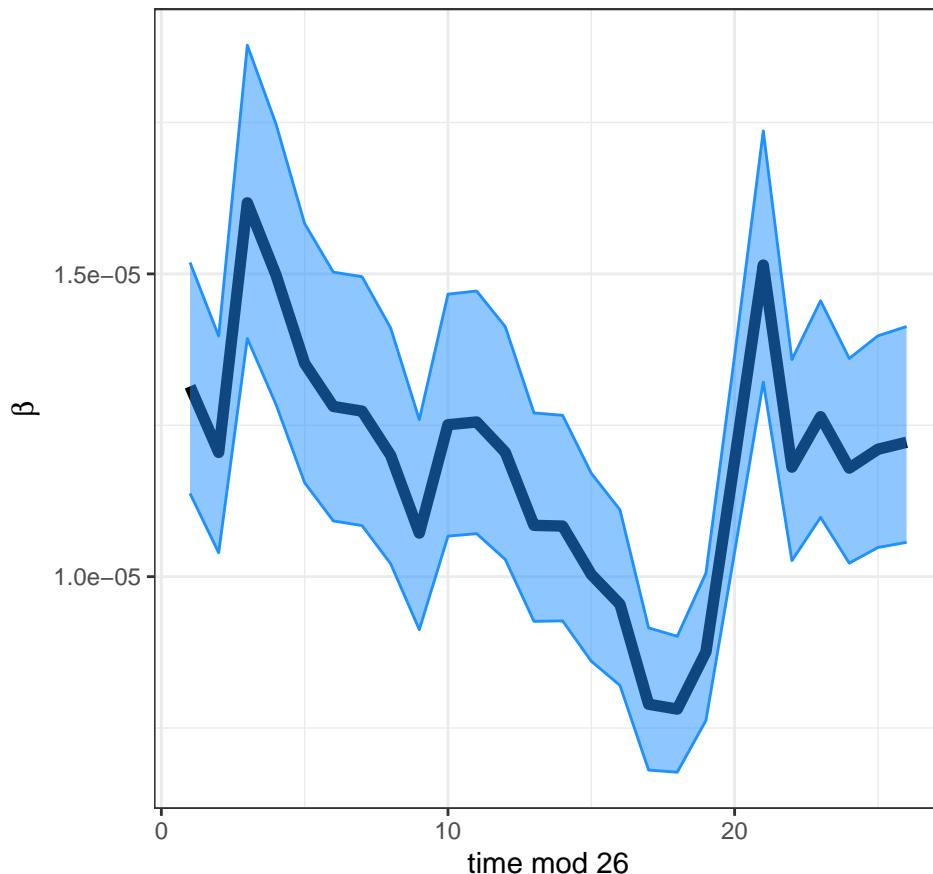
terms[c(252:299)] = bh
terms[c(300:307)] = bl
terms[c(308:355)] = bh
terms[c(356:364)] = bl
terms[c(1:6)] = bl
terms[c(7:99)] = bh
terms[c(100:115)] = bl
terms[c(116:199)] = bh
terms[c(200:251)] = bl

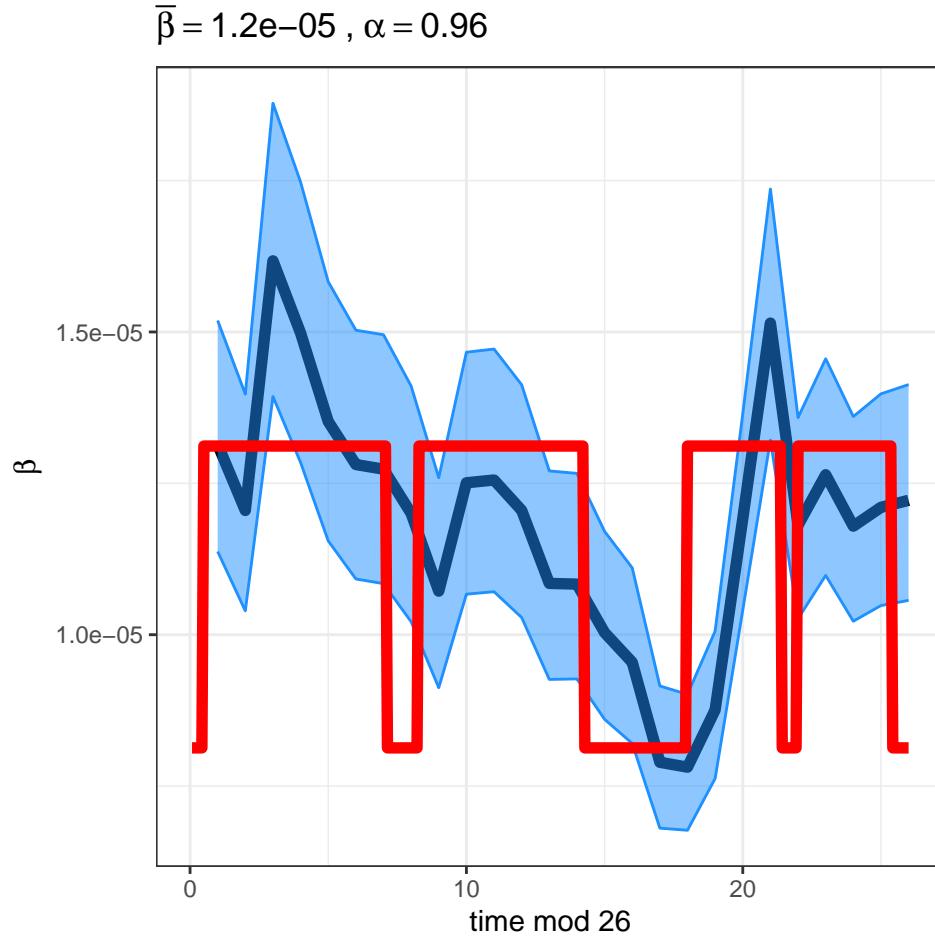
return(terms)
}

plotbeta(LondonRes) + annotate('line',x=seq(1,364,1)/14,y=mk_terms(mean(LondonRes$beta),0.21),lwd=2,col-

```

$$\bar{\beta} = 1.2 \times 10^{-5}, \alpha = 0.96$$





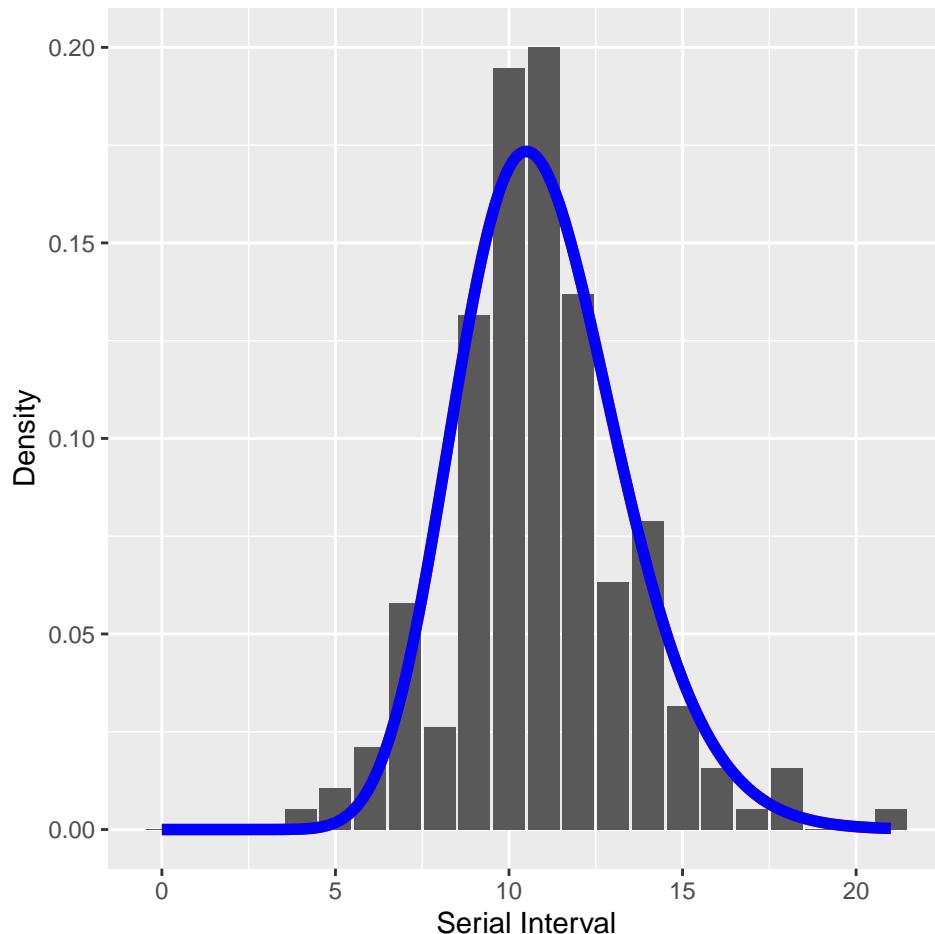
Important to note that the estimated transmission parameters from the *TSIR* model **do not** translate directly to the transmission parameters for continuous-time (ordinary differential or stochastic) models. The qualitative pattern is informative rather than the specific estimates. (Conceptually I would consider these estimates to be closer to reproduction numbers than transmission rates as they are usually presented.). Note also that the α used in the *TSIR* model is not an amplitude of seasonality but a correction (i.e. fudge) factor for the density dependence of the transmission term which—in some sense—can be used to adjust for artifacts arising from the discrete time approximation.

We also provided you with [Hope-Simpson's estimates](#) of the serial interval of measles and illustrated that it suggests that the latent and infectious periods of measles are less dispersed (less variable) than exponential and well described by a gamma distribution where the shape and scale parameters are approximately equal:

```
household <- as_tibble(read.csv('Materials/BYOM/Data/measles_hope_simpson.csv'))

SIR_gamma <- tibble(interval=seq(0,21,0.1),value=dgamma(seq(0,21,0.1),22,22/11))
SEIR_gamma <- tibble(interval=seq(0,21,0.1),value=dgamma(seq(0,21,0.1),22,22/11))

ggplot(household,aes(x=interval,y=B/sum(B))) + geom_col() +
  annotate('line',x=SIR_gamma$interval,y=SIR_gamma$value,col='red',lwd=2) +
  annotate('line',x=SEIR_gamma$interval,y=SEIR_gamma$value,col='blue',lwd=2) +
  xlab('Serial Interval') + ylab('Density')
```



Show: Standard *SEIR* (exponential) with sinusoidal forcing on P268

Show: Standard *SEIR* (exponential) with term-time forcing on P277

Show: Gamma *SEIR* with term-time forcing on P286

Chapter 13

Outbreak of influenza in a boarding school

Andrew Conlan (ajkc2@cam.ac.uk)

13.1 Data summary and challenge

In this practical you will calibrate a transmission model to describe the outbreak of influenza in a English boarding school.

For the purposes of this practical assume that the numbers of pupils in bed measure the numbers infectious on each date. Not a great assumption as they are to an extent self-isolating, but also not terrible given the relatively short latent and (effective) infectious period for influenza (~ 1 day).

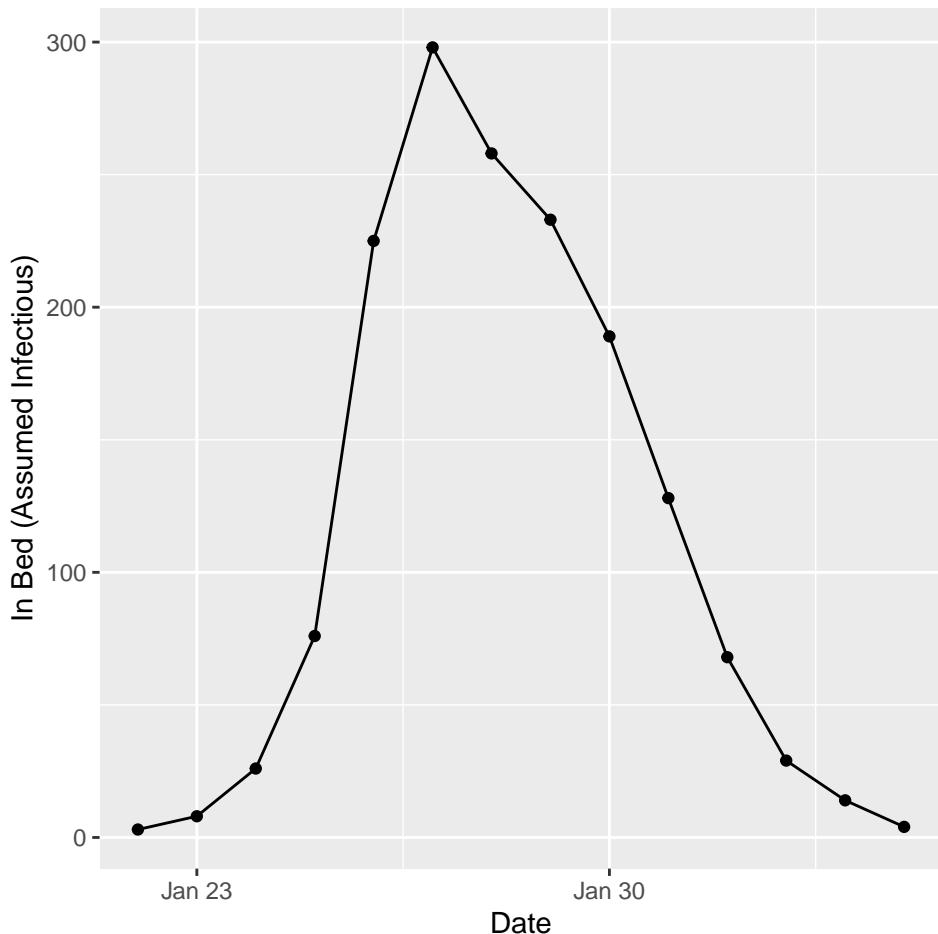
Model the likely impact on the outbreak should 80% of the boys have been vaccinated before the start of the outbreak with a vaccine with 50% (direct) protection from infection.

Note

For a single outbreak it is reasonable to neglect the potential for loss of immunity to reinfection so modelling vaccination in this case only amounts to changing the **initial conditions** of the model (i.e. the number susceptible and recovered when infection is introduced.) We are told to assume an exposed period of 1 hour and an infectious period of 2 days so a standard *SEIR* compartmental model should be sufficient. You should use a deterministic model for your initial calibration and then explore how the dynamics differ for the corresponding stochastic model.

```
require(outbreaks)
require(tidyverse)
require(deSolve)

ggplot(influenza_england_1978_school,aes(x=date,y=in_bed)) + geom_point() + geom_line() + xlab('Date') +
```



```
targetI <- influenza_england_1978_school$in_bed
```

Show: Deterministic model on P295

Show: Stochastic model on P298

Part VII

Appendices

References

- Anderson, R. M., and R. M. May. 1991. *Infectious Diseases of Humans*. Oxford University Press.
- Andrieu, Christophe, Arnaud Doucet, and Roman Holenstein. 2010. "Particle Markov Chain Monte Carlo Methods." *Journal of the Royal Statistical Society, Series B (Methodological)* 72 (3): 269–342.
- Anonymous. 1978. "Influenza in a Boarding School." *British Medical Journal* 1: 578.
- de Vries, Gerda, Thomas Hillen, Mark Lewis, Johannes Mueller, and Birgitt Schöenfisch. 2006. *A Course in Mathematical Biology: Quantitative Modeling with Mathematical and Computational Methods*. Society for Industrial; Applied Mathematics.
- Funk, Sebastian, Adam J. Kucharski, Anton Camacho, Rosalind M. Eggo, Laith Yakob & Lawrence M. Murray, and W. John Edmunds. 2016. "Comparative Analysis of Dengue and Zika Outbreaks Reveals Differences by Setting and Virus." *PLoS Neglected Tropical Diseases* 10 (12): e0005173.
- Gordon, N. J., D. J. Salmond, and A. F. M. Smith. 1993. "Novel Approach to Nonlinear/Non-Gaussian Bayesian State Estimation." *Radar and Signal Processing, IEE Proceedings F*. 140 (2): 107–13. <https://doi.org/10.1049/ip-f-2.1993.0015>.
- Greenhalgh, David, and Klaus Dietz. 1994. "Some Bounds on Estimates for Reproductive Ratios Derived from the Age-Specific Force of Infection." *Mathematical Biosciences* 124 (1): 9–57. [https://doi.org/10.1016/0025-5564\(94\)90023-X](https://doi.org/10.1016/0025-5564(94)90023-X).
- Hens, Niel, Ziv Shkedy, Marc Aerts, Christel Faes, Pierre Van Damme, and Philippe Beutels. 2012. *Modeling Infectious Disease Parameters Based on Serological and Social Contact Data*. Springer. <https://doi.org/10.1007/978-1-4614-4072-7>.
- Ionides, E. L., C. Bretó, and A. A. King. 2006. "Inference for Nonlinear Dynamical Systems." *Proceedings of the National Academy of Sciences USA* 103: 18438–43.
- Murray, J. D. 2003. *Mathematical Biology i - an Introduction*. 3rd ed. Springer.

Answers

Solution 1

The answer is, of course, 42.

[Return to task on P9](#)

Solution 2

R returns an error because you've asked the `seq()` function to move from 10 to 0 by adding 0.5 each time, which it clearly can't do. The step size would have to be -0.5 in order to generate decreasing numbers.

[Return to task on P25](#)

Solution 3

```
## 1.  
seq(2, 30, by = 2)  
## 2.  
seq(-2.5, 15.34, length.out = 14)  
## 3.  
seq(0, by = 0.04, length.out = 7)  
## 4.  
seq(101, -20, by = -11)
```

[Return to task on P26](#)

Solution 4

```
## 1.  
1:11  
## 1. (alternative)  
seq(1, 11, by = 1)  
## 2.  
seq(2, 50, by = 2)  
## 3.  
seq(17, 33, by = 2)  
## 4.  
c("M", "T", "W", "T", "F", "S", "S")
```

[Return to task on P31](#)

Solution 5

```
## generate a vector x of five elements from 1 to 5
x <- 1:5

## extract the third element of x
x[3]

## extract the 2, 3 and 4th elements of x
x[c(2, 3, 4)]

## change the first element of x to 17
x[1] <- 17

## extract all elements of x except the third
x[-3]

## extract all elements of x except the third
## and save to new object w
w <- x[-3]

## create a vector of length 5 called y with
## corresponding entries below
y <- c(1, 5, 2, 4, 7)

## extract all elements of y except the third
## and fifth
y[-c(3, 5)]

## extract elements 1, 4 and 5 from y
y[c(1, 4, 5)]

## create a vector i containing three elements:
## 1, 2 and 3
i <- 1:3

## extract elements 1, 2 and 3 from y
y[i]

## create a vector z of length 3 with elements
## 9, 10 and 11
z <- c(9, 10, 11)

## replace elements 1, 2 and 3 in y with z
y[i] <- z
```

[Return to task on P32](#)

Solution 6

```
## 1. this sorts the vector y into increasing order.
order(y)
## 2. this reverses the order of y.
rev(y)
## 3. this sorts the vector y into increasing order.
sort(y)

## there are various ways to sort `y` into decreasing order e.g.
sort(y, decreasing = TRUE)
rev(sort(y))
order(-y)
```

[Return to task on P32](#)

Solution 7

```
## creates a vector y containing entries
## 1, 2, ..., 10
y <- 1:10

## squares each entry of y
y^2

## takes the natural log of each entry of y
log(y)

## takes the anti-log of each entry of y
exp(y)

## creates a vector called x containing
## entries below
x <- c(5, 4, 3, 2, 1, 5, 4, 3, 2, 1)

## adds each element of x to its corresponding
## element in y
x + y

## multiplies each element of x with its corresponding
## element in y
x * y

## creates a vector of length 3 called z
## with entries below
z <- c(-1, 2.2, 10)

## repeats z 4 times, and takes the first
## 11 entries, which it then adds to
## each corresponding entry of y
z + y

## repeats z 4 times, and takes the first
## 11 entries, which it then multiplies with
## each corresponding entry of y
z * y
```

[Return to task on P33](#)

Solution 8

```

## create a vector x containing elements 1:10
x <- 1:10

## create a vector y containing the elements below
y <- c(5, 4, 3, 2, 1, 5, 4, 3, 2, 1)

## generate a logical vector where each
## element denotes whether x is < 4 (TRUE)
## or >= 4 (FALSE)
x < 4

## extracts all elements of x that are < 4
x[x < 4]

## extracts all elements of y where
## x < 4
y[x < 4]

## creates a logical vector where elements are TRUE
## if the element of y is > 1 AND <= 4, and FALSE
## otherwise
y > 1 & y <= 4

## extracts all elements of y that are > 1 and
## <= 4
y[y > 1 & y <= 4]

## extracts all elements of y that are not equal
## to 3 and pass them into a new vector z
z <- y[y != 3]

```

[Return to task on P34](#)

Solution 9

```
## Here are the steps:  
  
## 1. First create a sequence of numbers  
## from 1 to 10 in increments of 0.5  
## returns a vector  
seq(1, 10, by = 0.5)  
  
## 2. take the natural log of each  
## element of the vector - returns a vector  
log(seq(1, 10, by = 0.5))  
  
## 3. take the square root of each logged  
## element - returns a vector  
sqrt(log(seq(1, 10, by = 0.5)))  
  
## 4. sums all the elements of the  
## transformed vector - returns a single  
## number (a vector of length 1)  
sum(sqrt(log(seq(1, 10, by = 0.5))))
```

[Return to task on P35](#)

Solution 10

```

## create a vector y containing the numbers below
y <- c(5, 4, 3, 2, 10, 5, 4, 3, 2, 1)

## visualise y as a factor
factor(y)

## visualise y as a factor but set the levels
## this means some levels will have zero
## entries
factor(y, levels = 1:10)

## create a character vector y containing the numbers
## below coded as strings
y <- as.character(c(5, 4, 3, 2, 10, 5, 4, 3, 2, 1))

## visualise y as a factor - sorts levels
## lexicographically (hence 10 coded before 2 etc.)
factor(y)

## visualise y as a factor, but set levels that would
## be equivalent to if y was numeric instead of
## a character vector
factor(y, levels = sort(unique(as.numeric(y)))) 

## create a character vector y containing the strings below
y <- c("low", "mid", "mid", "high", "low")

## visualise y as a factor - sorts levels
## lexicographically (so alphabetically in this case)
factor(y)

## visualise y as a factor - sorts levels
## into a more sensible ordering
factor(y, levels = c("low", "mid", "high"))

```

[Return to task on P36](#)

Solution 11

```

## creates a (2 x 2) matrix with entries
## 1:4 placed down the columns first
matrix(1:4, 2, 2)

## creates a (2 x 2) matrix with entries
## 1:4 placed across the rows first
matrix(1:4, 2, 2, byrow = TRUE)

## creates a (3 x 4) matrix filled with zeros
matrix(0, 3, 4)

```

[Return to task on P38](#)

Solution 12

```
## create a vector vals and fill with values
vals <- c(1, 2, 3, 4, 5, 0.5, 2, 6, 0, 1, 1, 0)

## create a (4 x 3) matrix and fill with entries
## from vals, filling down the columns first
mat <- matrix(vals, 4, 3)

## extract the entry of mat in the
## 2nd row and 3rd column
mat[2, 3]

## extract the first row of mat
mat[1, ]

## extract the third column of mat
mat[, 3]

## extract all rows of mat except the second
mat[-2, ]

## extract entries in the first and third row,
## and second and third column
mat[c(1, 3), c(2, 3)]
```

[Return to task on P39](#)

Solution 13

```
## create a vector x1 containing the entries 1:3
x1 <- 1:3

## create a vector x2 containing the entries below
x2 <- c(7, 5, 6)

## create a vector x3 containing the entries below
x3 <- c(12, 19, 25)

## bind x1, x2 and x3 together as columns
## to form a (3 x 3) matrix
cbind(x1, x2, x3)

## bind x1, x2 and x3 together as rows
## to form a (3 x 3) matrix
rbind(x1, x2, x3)
```

[Return to task on P39](#)

Solution 14

```
## create a (3 x 3) matrix x containing
## the numbers 1:9 down the columns
x <- matrix(1:9, 3, 3)

## print x to the screen
x

## multiply each element of x by 2
x * 2

## multiply row 1 by 1, row 2 by 2 and row 3 by 3
x * c(1:3)
```

[Return to task on P40](#)

Solution 15

```
## convert columns to factors
ff$partners <- factor(ff$partners)
ff$partner.type <- factor(ff$partner.type)

## produce summary
summary(ff)
```

	partners	partner.type	longevity	thorax
##	0:25	Control	:25	Min. :16.00 Min. :0.640
##	1:50	Inseminated	:50	1st Qu.:46.00 1st Qu.:0.760
##	8:50	Virgin	:50	Median :58.00 Median :0.840
##				Mean :57.44 Mean :0.821
##				3rd Qu.:70.00 3rd Qu.:0.880
##				Max. :97.00 Max. :0.940

Notice that the summary now produces counts for each level of the relevant categorical variables.

[Return to task on P46](#)

Solution 16

Prints numbers in order.

```
## [1] "1 1"
## [1] "2 2"
## [1] "3 3"
## [1] "4 4"
## [1] "5 5"
## [1] "6 6"
## [1] "7 7"
## [1] "8 8"
## [1] "9 9"
## [1] "10 10"
```

[Return to task on P52](#)

Solution 17

Prints numbers in reverse order according to whatever is in sample **b**.

```
## [1] "10 5"
## [1] "9 9"
## [1] "8 7"
## [1] "7 8"
## [1] "6 3"
## [1] "5 10"
## [1] "4 1"
## [1] "3 6"
## [1] "2 4"
## [1] "1 2"
```

[Return to task on P52](#)

Solution 18

Prints numbers in forward order according to whatever is in vector **b**.

```
## [1] "1 2"
## [1] "2 4"
## [1] "3 6"
## [1] "4 1"
## [1] "5 10"
## [1] "6 3"
## [1] "7 8"
## [1] "8 7"
## [1] "9 9"
## [1] "10 5"
```

[Return to task on P52](#)

Solution 19

Prints numbers in a random order from whatever is in vector **b**.

```
## [1] "4 1"
## [1] "6 3"
## [1] "8 7"
## [1] "9 9"
## [1] "1 2"
## [1] "7 8"
## [1] "5 10"
## [1] "3 6"
## [1] "10 5"
## [1] "2 4"
```

[Return to task on P52](#)

Solution 20

```
for (i in 1:12) {
  result <- i * 12
  print(paste(i, "times 12 is", result))
}

## [1] "1 times 12 is 12"
## [1] "2 times 12 is 24"
## [1] "3 times 12 is 36"
## [1] "4 times 12 is 48"
## [1] "5 times 12 is 60"
## [1] "6 times 12 is 72"
## [1] "7 times 12 is 84"
## [1] "8 times 12 is 96"
## [1] "9 times 12 is 108"
## [1] "10 times 12 is 120"
## [1] "11 times 12 is 132"
## [1] "12 times 12 is 144"
```

[Return to task on P53](#)

Solution 21

```
i <- 1
while(i <= 12) {
  result <- i * 12
  print(paste(i, "times 12 is", result))
  i <- i + 1
}

## [1] "1 times 12 is 12"
## [1] "2 times 12 is 24"
## [1] "3 times 12 is 36"
## [1] "4 times 12 is 48"
## [1] "5 times 12 is 60"
## [1] "6 times 12 is 72"
## [1] "7 times 12 is 84"
## [1] "8 times 12 is 96"
## [1] "9 times 12 is 108"
## [1] "10 times 12 is 120"
## [1] "11 times 12 is 132"
## [1] "12 times 12 is 144"
```

[Return to task on P53](#)

Solution 22

```
runningSum <- 0
for(i in 1:15) {
  runningSum <- runningSum + 2^i
}
print(paste("Sum is", runningSum))
```

[1] "Sum is 65534"

Note you can do the above more efficiently using vectorised operators, e.g. `sum(2^(1:15))`.

[Return to task on P53](#)

Solution 23

```
thisIDX <- 1
thisPower <- 2^thisIDX
while(thisPower < 1000000) {
  thisIDX <- thisIDX + 1
  thisPower <- 2^thisIDX
}
print(paste("2^", thisIDX, "=", thisPower, sep = ""))
```

[1] "2^20=1048576"

[Return to task on P54](#)

Solution 24

```
sayHelloWithArg()      # error, since not passed argument
```

Error in sayHelloWithArg(): argument "whoTo" is missing, with no default

```
# echos function's code to screen (happens in general if
# type function's name without brackets)
sayHelloWithArg
```

```
## function(whoTo) {
##   print(paste("Hello", whoTo))
## }
```

```
sayHelloWithArg("nik")  # works as expected
```

[1] "Hello nik"

```
sayHello("nik")        # error, since too many arguments for the simpler function
```

Error in sayHello("nik"): unused argument ("nik")

```
sayHelloWithArg(nik)    # error, since variable nik is not defined
```

```
## Error in eval(expr, envir, enclos): object 'nik' not found

sayHelloWithArg(whoto="nik") # error, since argument names are case sensitive

## Error in sayHelloWithArg(whoto = "nik"): unused argument (whoto = "nik")

sayHelloWithArg(whoTo="nik") # works as expected

## [1] "Hello nik"
```

[Return to task on P54](#)

Solution 25

The line `ghost <- buster(canary, -cat)` sets `ghost` to be 2, but leaves `cat` and `canary` unchanged.

[Return to task on P55](#)

Solution 26

```
# my function to sum elements of a vector and return that and the mean
mySumAndMean <- function(toSum) {
  retVal <- 0
  for(i in 1:length(toSum)) {
    retVal <- retVal + toSum[i]
  }
  meanVal <- retVal/length(toSum) # find the mean, too
  return(c(retVal,meanVal))      # return two values by wrapping up into vector
}
```

[Return to task on P56](#)

Solution 27

There is a mistake in the code on line 8. The condition in the `if` statement should be checking for `FALSE`, not `TRUE`.

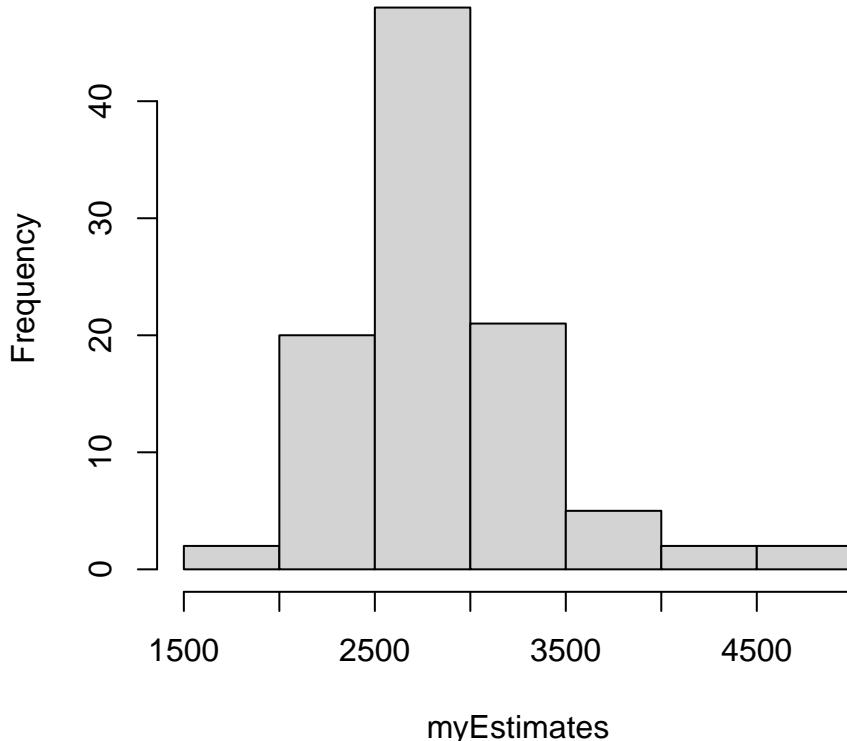
[Return to task on P57](#)

Solution 28

```

myEstimates <- numeric(0)
numRuns <- 100
for(i in 1:numRuns) {
  totStickers <- 427
  gotSticker <- rep(FALSE,totStickers)
  numLeft <- totStickers
  numStickersBought <- 0
  while(numLeft > 0) {
    numStickersBought <- numStickersBought + 1
    thisSticker <- sample(1:totStickers, 1)
    if(gotSticker[thisSticker] == FALSE) {      # this is the line which had an error (fixed here)
      gotSticker[thisSticker] <- TRUE
      numLeft <- numLeft - 1
    }
  }
  myEstimates[i] <- numStickersBought
}
hist(myEstimates)

```

Histogram of myEstimates

```
mean(myEstimates)
```

```
## [1] 2738.61
```

[Return to task on P57](#)

Solution 29

The given function returns N iterates of the logistic map starting at x_0 (note, it does not store the initial value).

[Return to task on P58](#)

Solution 30

As you change the value of r in the above code the system goes from having a single equilibrium in the long term, to oscillating between a pair of values, to oscillating between four values. By $r = 3.7$, the trajectory depends very sensitively to the initial value x_0 (as is demonstrated by the below code). This is called deterministic chaos.

[Return to task on P59](#)

Solution 31

The given code performs N iterations of the logistic map starting from x_0 but does not store them. It then calculates, stores and returns the following M iterations of the logistic map.

[Return to task on P60](#)

Answer 32

The key idea is that, for each value of r , it calculates $N = 1000$ iterations of the map, but totally ignores them. It then plots the values of the next 100 iterations (as the y -value on a graph which has the value of r as the x -ordinate). The idea behind all this is to let the system equilibrate—if it is ever going to do so. So from the bifurcation diagram you can see from this picture that there is/are:

- a single equilibrium for $2.5 < r < 3.0$
- two equilibria for $3.0 < r < 3.45$ (-ish)
- four equilibria for 3.45 (-ish) $< r < 3.54$ (-ish)
- by $r \sim 3.57$, we have chaos (as seen in plots for different starting conditions above) which here corresponds to a more or less random—although lying within certain bounds—set of 100 points plotted for each r .

This is explained well on the relevant [Wikipedia page](#)

[Return to task on P61](#)

Solution 33

```
rand <- runif(1, 0, 1)
if (rand < 1/6) {
  print(1)
} else if (rand < 2/6) {
  print(2)
} else if (rand < 3/6) {
  print(3)
} else if (rand < 4/6) {
  print(4)
} else if (rand < 5/6) {
  print(5)
} else {
  print(6)
}
```

```
## [1] 3
```

[Return to task on P74](#)

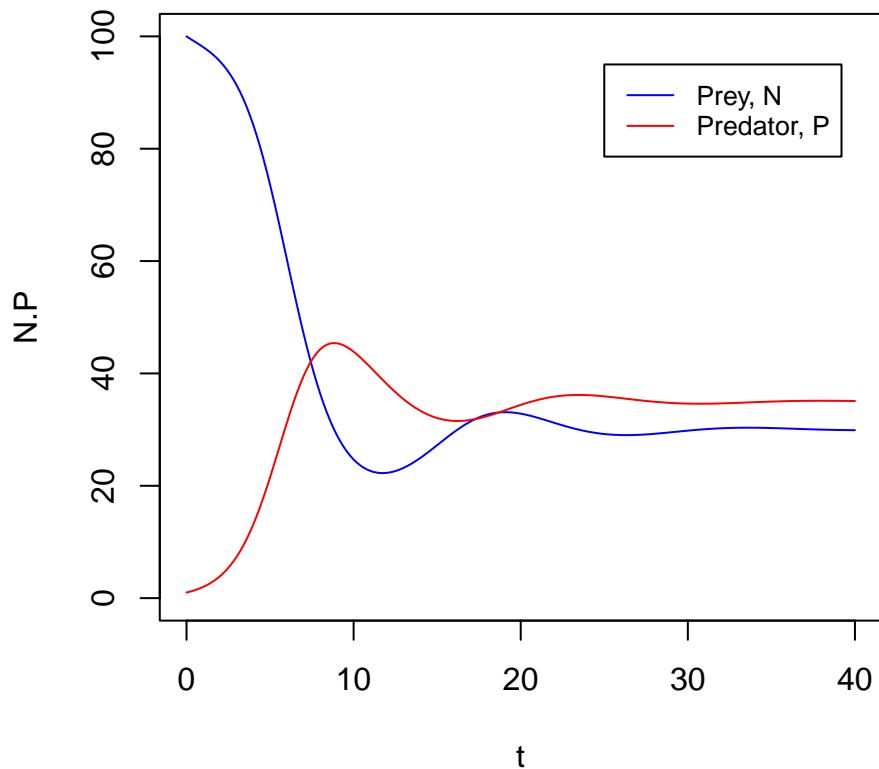
Solution 35

```
# plot the first line using `plot()`
plot(predation_sol[,1], predation_sol[,2], type = "l", main='Predation model',
      xlab='t', ylab='N.P', ylim = c(0, 100), col = "blue")

# plot the second line using `lines()`
lines(predation_sol[,1], predation_sol[,3], col = "red")

# add legend using `legend()`
legend(25, 95, legend=c("Prey, N", "Predator, P"),
       col=c("blue", "red"), lty=1, cex=0.8)
```

Predation model



Return to task on P85

Solution 36

```

plot(predation_sol[,2], predation_sol[,3], type = "l",
      main='Predation model', xlab='N', ylab='P',
      xlim = c(0, 100), ylim = c(0, 45), col = "black")

# N = 0
abline(v = 0, col = "blue")

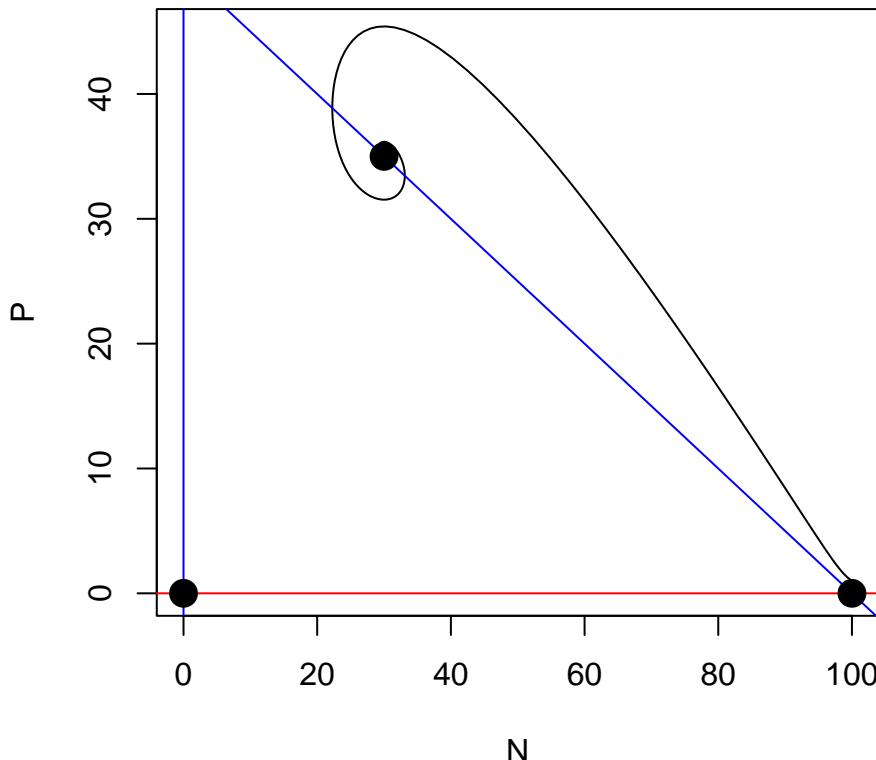
# P = 0
abline(h = 0, col = "red")

# P = (r/a) - (rN/Ka)
intercept = predation_par["r"]/predation_par["a"]
slope = -predation_par["r"]/(predation_par["K"]*predation_par["a"])

abline(intercept, slope, col = "blue")

# equilibrium points
points(0, 0, pch = 16, cex = 2)
points(predation_par["K"], 0, pch = 16, cex = 2)
points(
  x = predation_par["d"]/predation_par["b"],
  y = (predation_par["r"]/predation_par["a"]) -
    ((predation_par["r"]*predation_par["d"])/
      (predation_par["a"]*predation_par["b"]*predation_par["K"])),
  pch = 16,
  cex = 2
)

```

Predation model

[Return to task on P85](#)

Solution 37

- Only if you started at another equilibrium point, otherwise this equilibrium is globally stable and will attract all trajectories.

```
2. predation_init <- c(N=1, P=100)

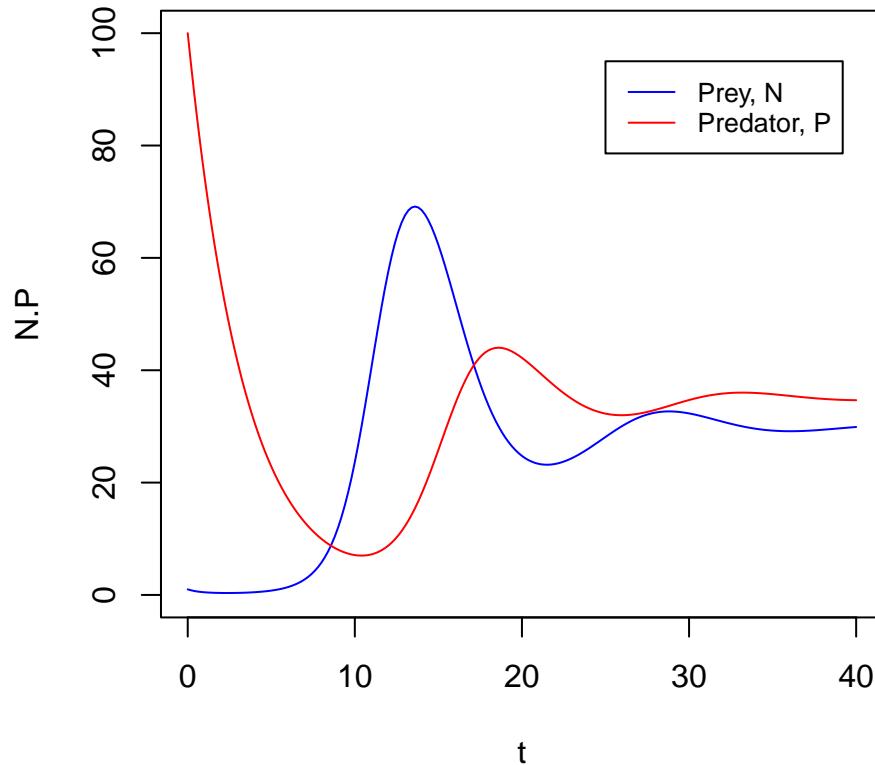
# Numerical solution
predation_sol <- lsoda(predation_init,
                        predation_t,
                        predation_dyn,
                        predation_par)

# plot the first line using `plot()`
plot(predation_sol[,1], predation_sol[,2], type = "l",
      main='Predation model', xlab='t', ylab='N.P',
      ylim = c(0, 100), col = "blue")

# plot the second line using `lines()`
lines(predation_sol[,1], predation_sol[,3], col = "red")

# add legend using `legend()`
legend(25, 95, legend=c("Prey, N", "Predator, P"),
       col=c("blue", "red"), lty=1, cex=0.8)
```

Predation model



[Return to task on P86](#)

Answer 38

Hopefully! No food for the predators but because this is deterministic, the populations still reach the equilibrium.

[Return to task on P86](#)

Answer 39

No stochastic die out.

[Return to task on P86](#)

Solution 40

In the below figure, $b = 0.02$ and $d = 0.01$. We can see that the prey population essentially dies out. This changes the stability of the equilibrium points so that the stable equilibrium is the one with both populations near 0.

```
predation_par <- c(a=0.01, b=0.02, d=0.3, K=100, r=1)
predation_init <- c(N=100, P=1)
predation_t <- seq(0,40,0.2)

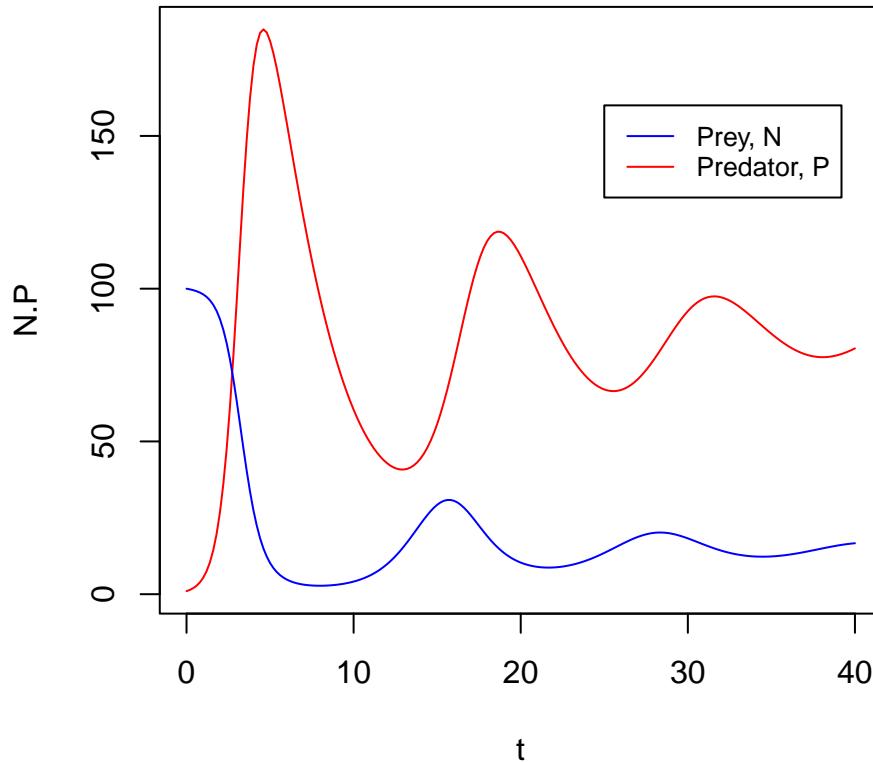
# Numerical solution
predation_sol <- lsoda(predation_init,
                        predation_t,
                        predation_dyn,
                        predation_par)

# plot the first line using `plot()`
plot(predation_sol[,1], predation_sol[,3], type = "l",
      main='Predation model', xlab='t', ylab='N.P', col = "red")

# plot the second line using `lines()`
lines(predation_sol[,1], predation_sol[,2], col = "blue")

# add legend using `legend()`
legend(25, 160, legend=c("Prey, N", "Predator, P"),
       col=c("blue", "red"), lty=1, cex=0.8)
```

Predation model



[Return to task on P86](#)

Solution 41

```
par(mfrow = c(2,2))

b_vec = c(0.01, 0.05, 0.1, 0.2)
d_vec = b_vec/(0.01/0.3)

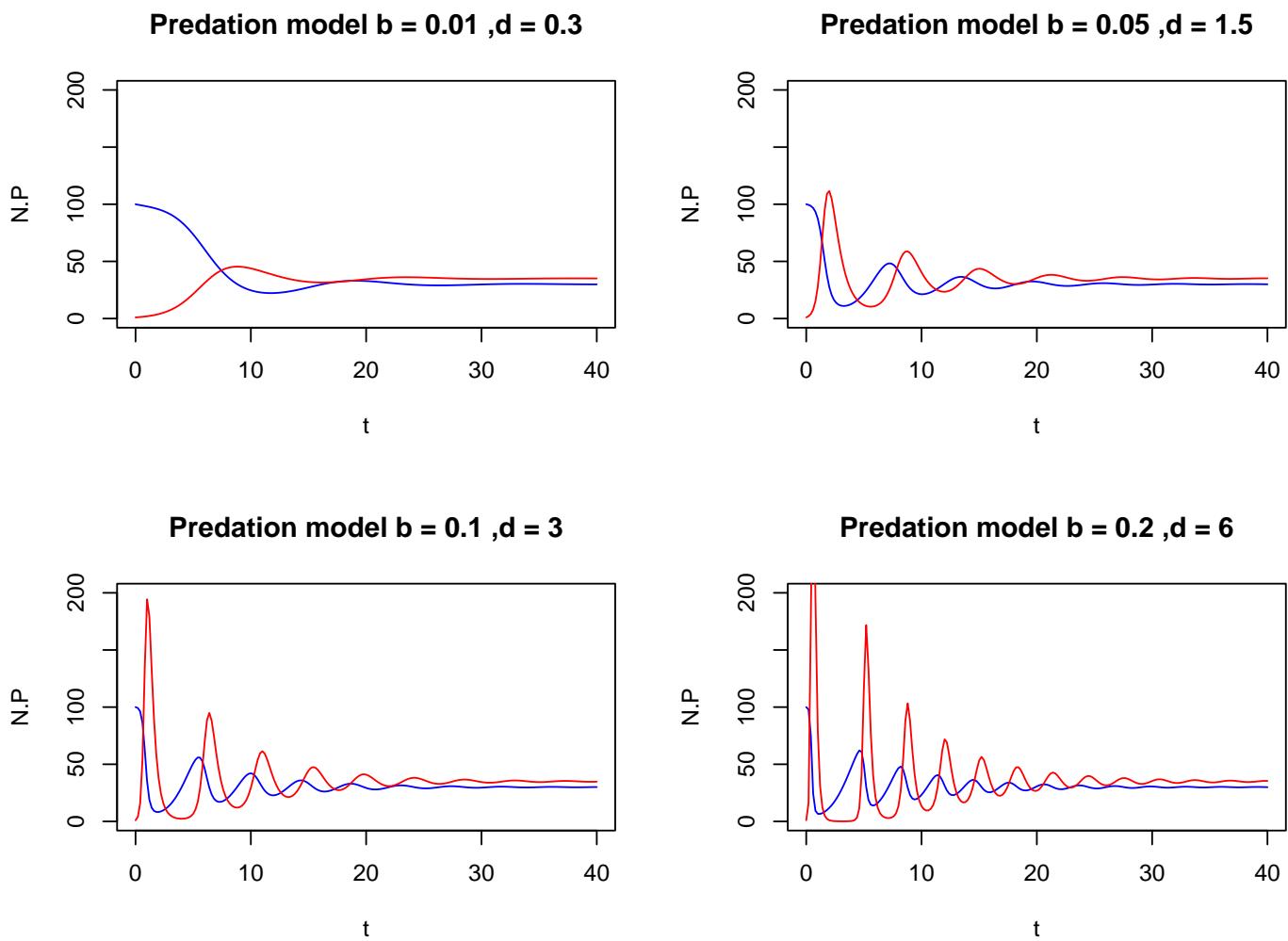
for(i in 1:4) {
  predation_par <- c(a=0.02, b=b_vec[i], d=d_vec[i], K=100, r=1)
  predation_init <- c(N=100, P=1)
  predation_t <- seq(0,40,0.2)

  # Numerical solution
  predation_sol <- lsoda(predation_init,
                          predation_t,
                          predation_dyn,
                          predation_par)

  plot(predation_sol[,1], predation_sol[,2], type = "l", col = "blue",
        main = paste("Predation model", "b =", b_vec[i], ",d =", d_vec[i]),
        xlab = "t", ylab = "N.P", ylim = c(0, 200))

  lines(predation_sol[,1], predation_sol[,3], col = "red")
}

par(mfrow = c(1, 1))
```



Similar equilibrium but it takes different times to reach it.

[Return to task on P86](#)

Solution 42

$$\begin{aligned}\frac{dS}{dt} &= -\frac{\beta SI}{N} \\ \frac{dI}{dt} &= \frac{\beta SI}{N} - \gamma I \\ \frac{dR}{dt} &= \gamma I\end{aligned}$$

[Return to task on P89](#)

Solution 43

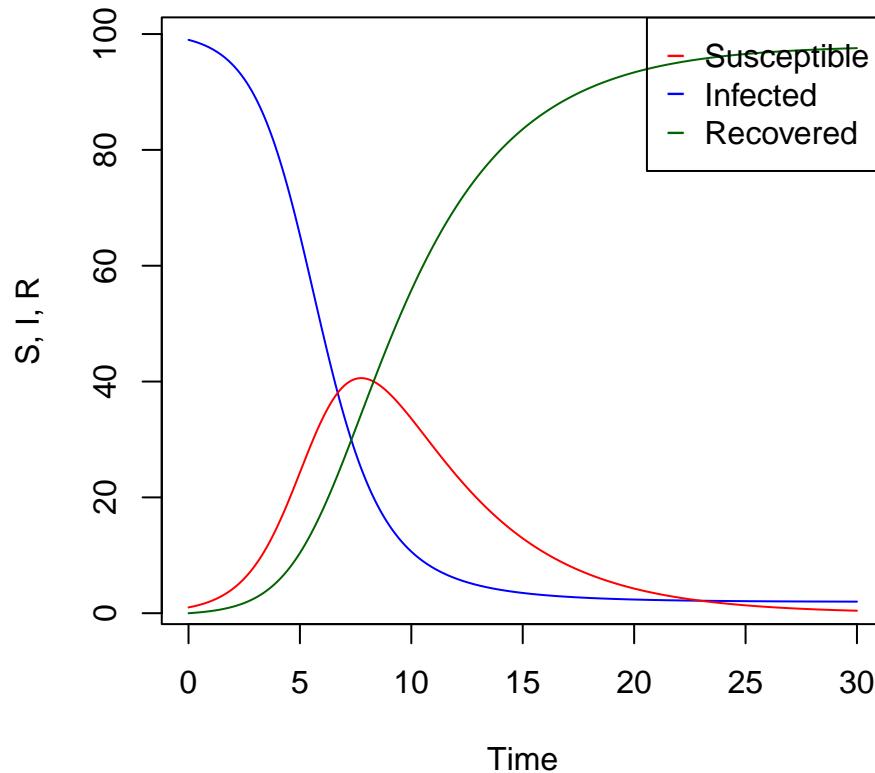
$$\begin{aligned}\frac{dN}{dt} &= \frac{dS}{dt} + \frac{dI}{dt} + \frac{dR}{dt} \\ &= \frac{-\beta SI}{N} + \frac{\beta SI}{N} - \gamma I + \gamma I \\ &= 0\end{aligned}$$

[Return to task on P89](#)

Solution 44

```
plot(TIME, S, col = "blue", type = "l", ylab = "S, I, R", xlab = "Time")
lines(TIME, I, col = "red")
lines(TIME, R, col = "darkgreen")

legend("topright", legend = c("Susceptible", "Infected", "Recovered"),
       col = c("red", "blue", "darkgreen"),
       pch = c("-", "-", "-"))
```



[Return to task on P90](#)

Solution 45

Approaching the steady state.

[Return to task on P92](#)

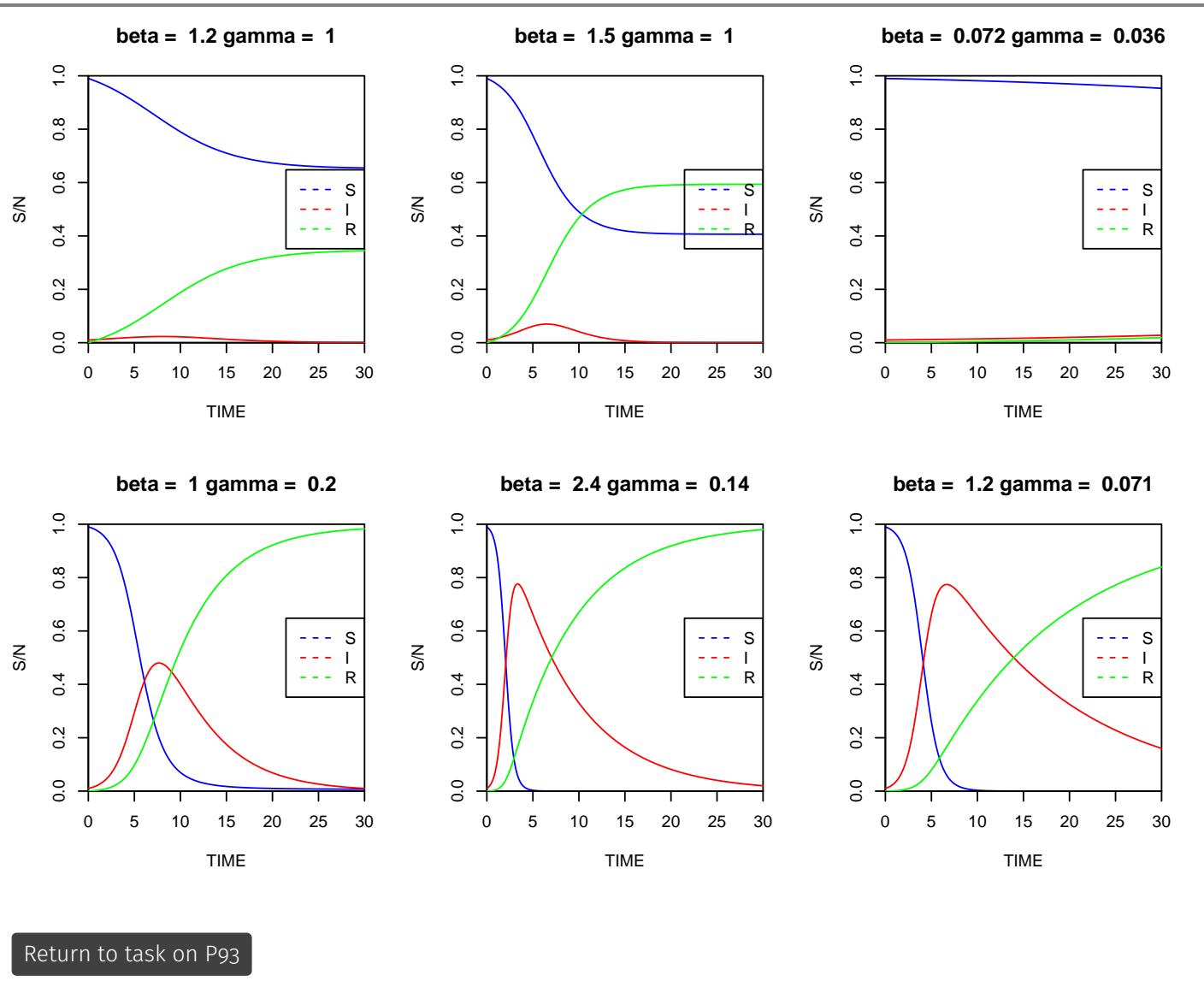
Solution 47

```

# set up a 2 by 3 grid for the plots:
par(mfrow=c(2,3), xaxs='i', yaxs='i')
# define the parameters:
infperiod <- c(1,1,28,5,7,14)
Rzero <- c(1.2,1.5,2,5,17,17)
SIR.t =seq(0,500, by=0.1)
for (i in 1:6) {
  gamma = signif(1/infperiod[i],2)
  beta = signif(Rzero[i]*gamma,2)
  SIR_par <- c(beta,gamma)
  SIR.sol <- lsoda(SIR_init,SIR_t,SIR_dyn,SIR_par)
  TIME <- SIR.sol[,1]
  S <- SIR.sol[,2]
  I <- SIR.sol[,3]
  R <- SIR.sol[,4]
  N <- S + I + R
  plot(TIME,S/N,col='blue', type='l', ylim=c(0,1),
    main=paste("beta = ",beta,"gamma = ",gamma))
  lines(TIME,I/N,col="red")
  lines(TIME,R/N,col="green")
  legend("right", col = c("blue","red","green"),legend = c("S","I","R"), lty = 2)
  print(paste("peak size = ",max(I/N)))
  print(paste("peak time = ",TIME[which.max(I/N)]))
  print(paste("Epidemic size = ",max(R/N)))
}
par(mfrow = c(1, 1))

## [1] "peak size =  0.0231066389917149"
## [1] "peak time =  8.1"
## [1] "Epidemic size =  0.344611420022074"
## [1] "peak size =  0.0697227613470483"
## [1] "peak time =  6.5"
## [1] "Epidemic size =  0.593574217769393"
## [1] "peak size =  0.0278724769684512"
## [1] "peak time =  30"
## [1] "Epidemic size =  0.0189539739504256"
## [1] "peak size =  0.480101096709578"
## [1] "peak time =  7.7"
## [1] "Epidemic size =  0.982652741440976"
## [1] "peak size =  0.776354567432811"
## [1] "peak time =  3.3"
## [1] "Epidemic size =  0.979907643827461"
## [1] "peak size =  0.774111948146292"
## [1] "peak time =  6.7"
## [1] "Epidemic size =  0.840041300875431"

```



[Return to task on P93](#)

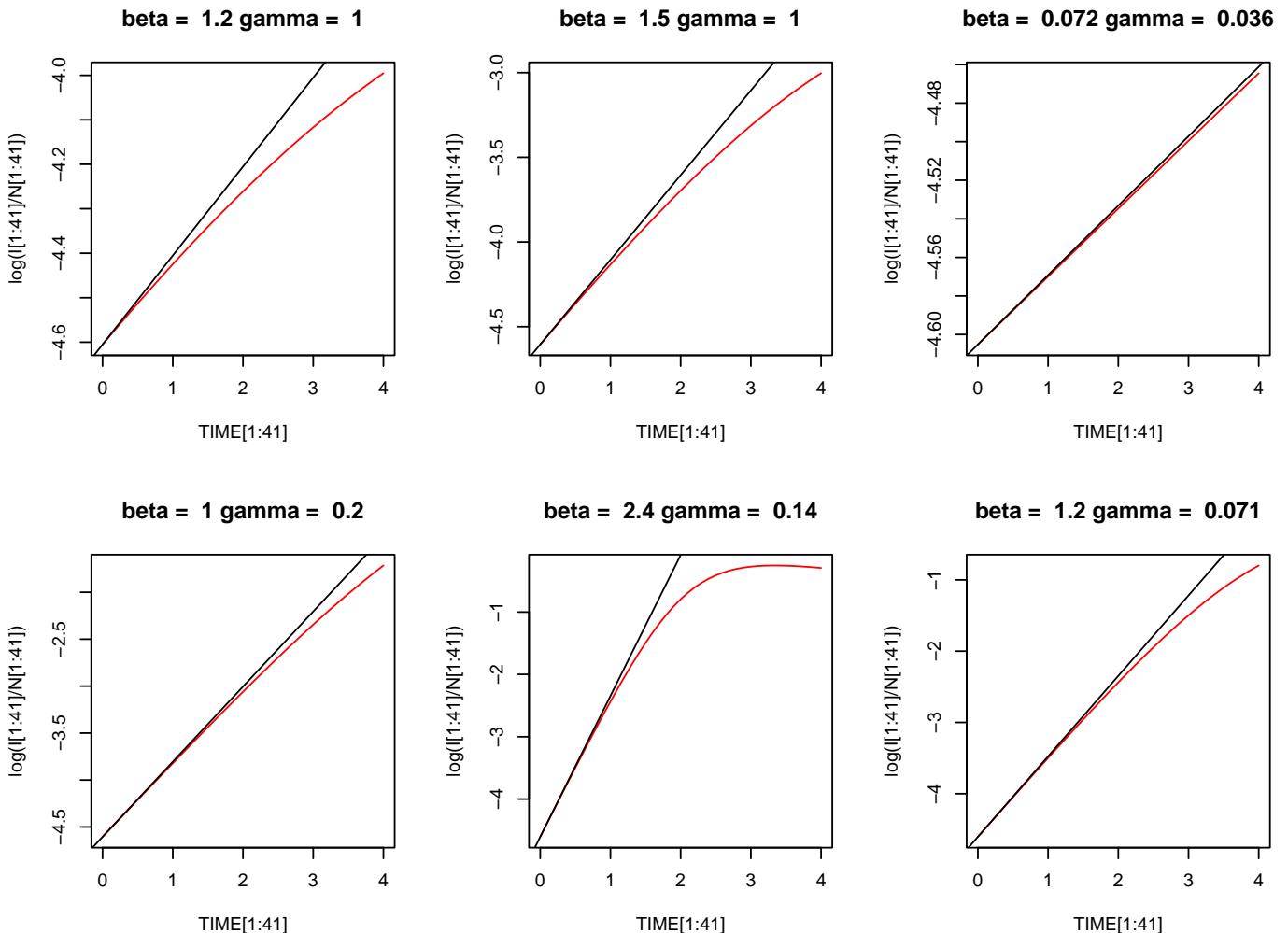
Solution 48

Peak size and epidemic size.

[Return to task on P93](#)

Solution 49

```
#checking initial gradient
par(mfrow=c(2,3))
# define the parameters:
infperiod <- c(1,1,28,5,7,14)
Rzero <- c(1.2,1.5,2,5,17,17)
SIR.t =seq(0,500, by=0.1)
for (i in 1:6) {
  gamma = signif(1/infperiod[i],2)
  beta = signif(Rzero[i]*gamma,2)
  SIR_par <- c(beta,gamma)
  SIR.sol <- lsoda(SIR_init,SIR_t,SIR_dyn,SIR_par)
  TIME <- SIR.sol[,1]
  S <- SIR.sol[,2]
  I <- SIR.sol[,3]
  R <- SIR.sol[,4]
  N <- S + I + R
  plot(TIME[1:41],log( I[1:41]/N[1:41] ),col='red', type='l',
       main=paste("beta = ",beta,"gamma = ",gamma))
  abline(a = log( I[1]/N[1] ), b = beta-gamma, col = "black")
}
par(mfrow=c(1,1))
```



[Return to task on P94](#)

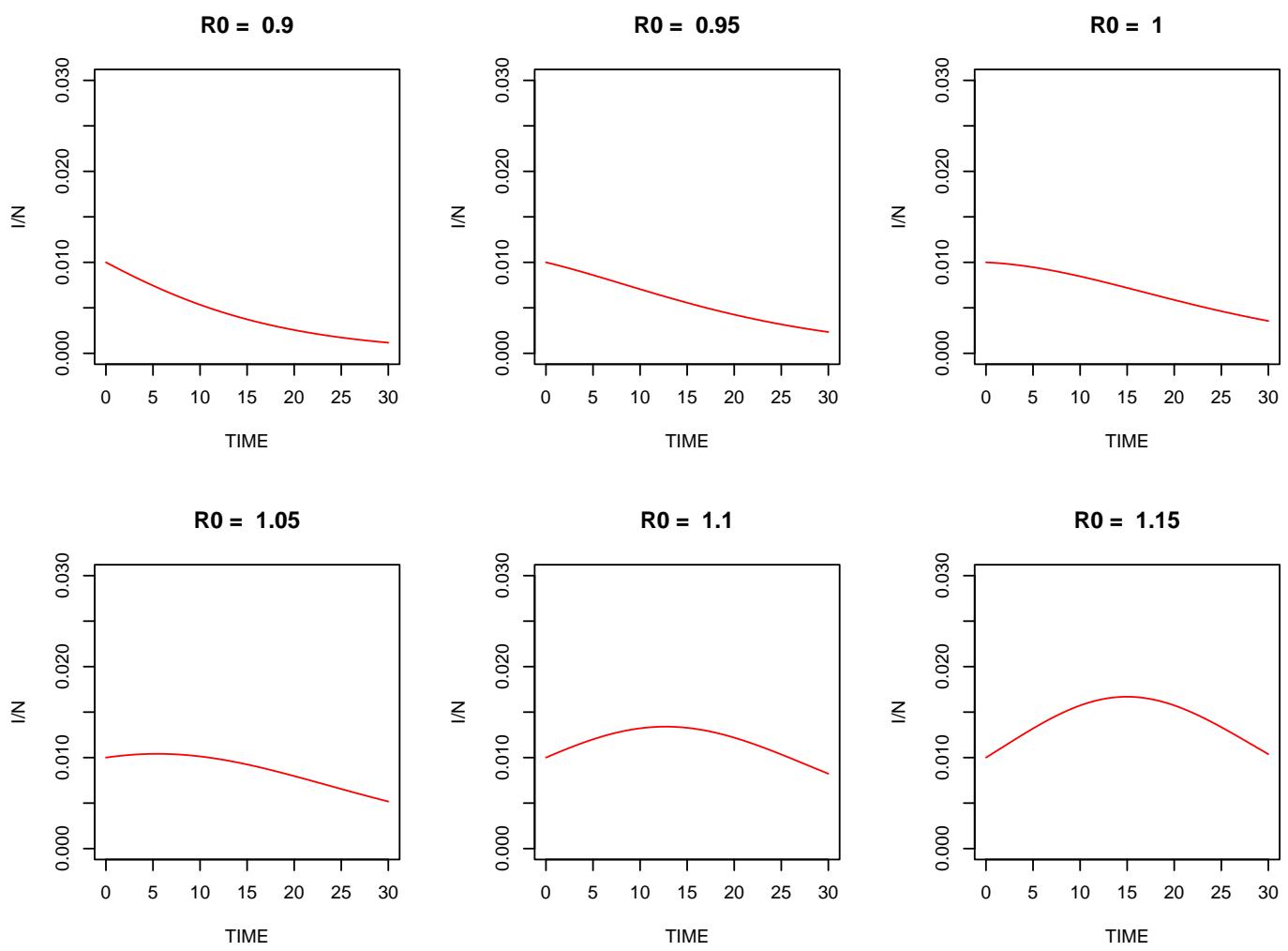
Solution 50

As our initial conditions are with 99 susceptible people, and $R_0 = 1$ this renders the **effective reproduction number** less than 1 so $I(t)$ decreases.

[Return to task on P94](#)

Solution 51

```
### looking around R0 = 1
par(mfrow=c(2,3))
# define the parameters:
SIR.init <- c(99,1,0)
infperiod <- rep(2,6)
Rzero <- c(0.9,0.95,1,1.05,1.1, 1.15)
SIR.t =seq(0,100, by=0.1)
for (i in 1:6) {
  gamma<- signif(1/infperiod[i],2)
  beta <- signif(Rzero[i]*gamma,2)
  SIR_par <- c(beta,gamma)
  SIR.sol <- lsoda(SIR_init,SIR_t,SIR_dyn,SIR_par)
  TIME <- SIR.sol[,1]
  S <- SIR.sol[,2]
  I <- SIR.sol[,3]
  R <- SIR.sol[,4]
  N <- S + I + R
  plot(TIME, I/N,col='red', type='l', ylim=c(0,0.03),
  main=paste("R0 = ",Rzero[i]))
}
par(mfrow=c(1,1))
```



[Return to task on P94](#)

Answer 52

The average infectious period is $1/\gamma = 5$ days.

[Return to task on P95](#)

Solution 53

```

##### Extra exercise
# I am putting everything in at once

SEITRS.dyn <- function(t, var, par) {
  # Rename the variables and parameters
  S <- var[1]
  E <- var[2]
  I <- var[3]
  tr <- var[4]
  R <- var[5]
  N <- S + E + I + tr + R
  beta <- par[1]
  gamma <- par[2]
  epsilon <- par[3] #rate of latency loss
  tau <- par[4] #treatment rate
  omega <- par[5] #recovery rate for treated people

  # Derivatives
  dS <- -beta*S*I/N
  dE <- beta*S*I/N - epsilon*E
  dI <- epsilon*E - gamma*I - tau*I
  dT <- tau*I - omega*tr
  dR <- gamma*I + omega*tr
  # Return the 5 values
  list(c(dS, dE, dI, dT, dR))
}

beta <- 2
gamma <- 0.25
epsilon <- 1
tau <- 1
omega <- gamma*2

SEITRS.par <- c(beta,gamma,epsilon,tau,omega)
SEITRS.init <- c(99,1,0,0,0)
SEITRS.t <- seq(0,30,by=0.1)

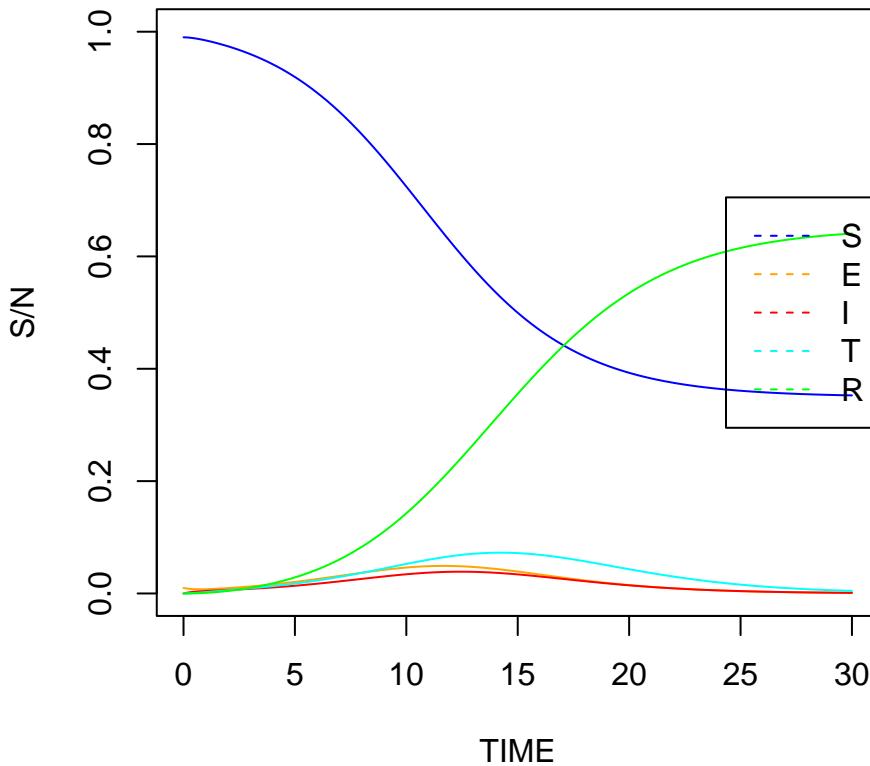
SEITRS.sol <- lsoda(SEITRS.init,SEITRS.t,SEITRS.dyn,SEITRS.par)

TIME <- SEITRS.sol[,1]
S <- SEITRS.sol[,2]
E <- SEITRS.sol[,3]
I <- SEITRS.sol[,4]
tr <- SEITRS.sol[,5]
R <- SEITRS.sol[,6]

N <- S + E + I + tr + R

plot(TIME,S/N,col='blue', type='l', ylim=c(0,1))
lines(TIME,E/N,col="orange")
lines(TIME,I/N,col="red")
lines(TIME,tr/N,col="cyan")
lines(TIME,R/N,col="green")
legend("right", col = c("blue","orange", "red","cyan","green"), legend = c("S","E","I","T","R"), lty = 2)

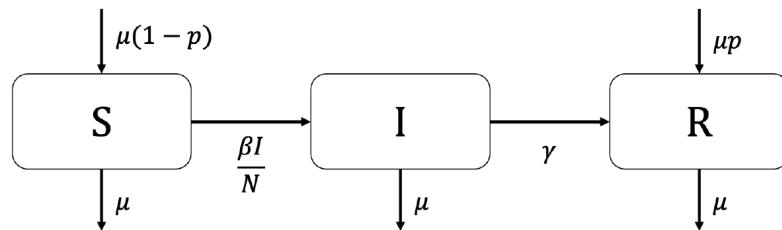
```



Plenty to note on the different dynamics as you experiment with the above- one thing to examine is that we now have multiple infected states and this can make calculating R_0 more complicated as we need to use the next generation matrix approach, shown later in the course.

[Return to task on P95](#)

Solution 54



[Return to task on P99](#)

Solution 55

$$\begin{aligned}\frac{dS}{dt} &= \mu(1-p)N - \frac{\beta SI}{N} - \mu \\ \frac{dI}{dt} &= \frac{\beta SI}{N} - \gamma I - \mu \\ \frac{dR}{dt} &= \mu p N + \gamma I - \mu\end{aligned}$$

[Return to task on P99](#)

Solution 56

```
library(deSolve)

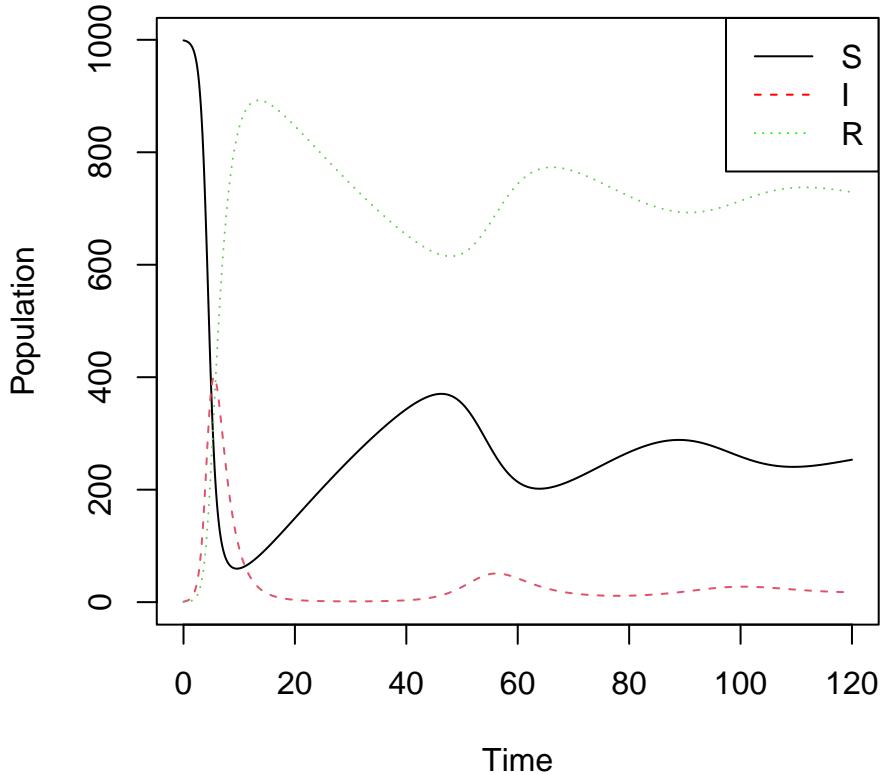
SIR.dyn <- function(t,var,par) {
  S <- var[1]
  I <- var[2]
  R <- var[3]
  N <- S+I+R
  beta <- par[1]
  gamma <- par[2]
  mu <- par[3]
  p <- par[4]

  dS <- (1-p)*mu*N - beta*S*I/N - mu*S
  dI <- beta*S*I/N - gamma*I - mu*I
  dR <- gamma*I - mu*R + p*mu*N
  return(list(c(dS,dI,dR)))
}

SIR.par<-c(beta=2,gamma=0.5,mu=1/70.0,p=0)
SIR.init<-c(S=999,I=1,R=0)
SIR.t<-seq(0,120,0.1)

det.sol <- lsoda(SIR.init,SIR.t,SIR.dyn,SIR.par)

matplot(det.sol[,1],det.sol[,2:4],type='l',xlab='Time',ylab='Population')
legend('topright',c('S','I','R'),col=c('black','red','green'),lty=c(1,2,3))
```



[Return to task on P100](#)

Solution 57

$$p_{vac} = 1 - \frac{1}{R_0} = 1 - \frac{\gamma + \mu}{\beta} = 26/35 \simeq 0.75$$

[Return to task on P100](#)

Solution 58

```

# Values of vaccination coverage to try
p_vals = seq(0.1,1.0,0.1)
# Color palette for plotting
col_vals = rainbow(length(p_vals))

# Make an empty plot
plot(0,xlim=c(0,150),ylim=c(0,500),pch=' ',xlab='Time',ylab='Population')

# Initial conditions and time points same for each solution
SIR.init<-c(S=999,I=1,R=0)
SIR.t<-seq(0,150,0.1)

for(i in 1:length(p_vals)) {
  # Change vaccination coverage
  SIR.par<-c(beta=2,gamma=0.5,mu=1/70.0,p=p_vals[i])

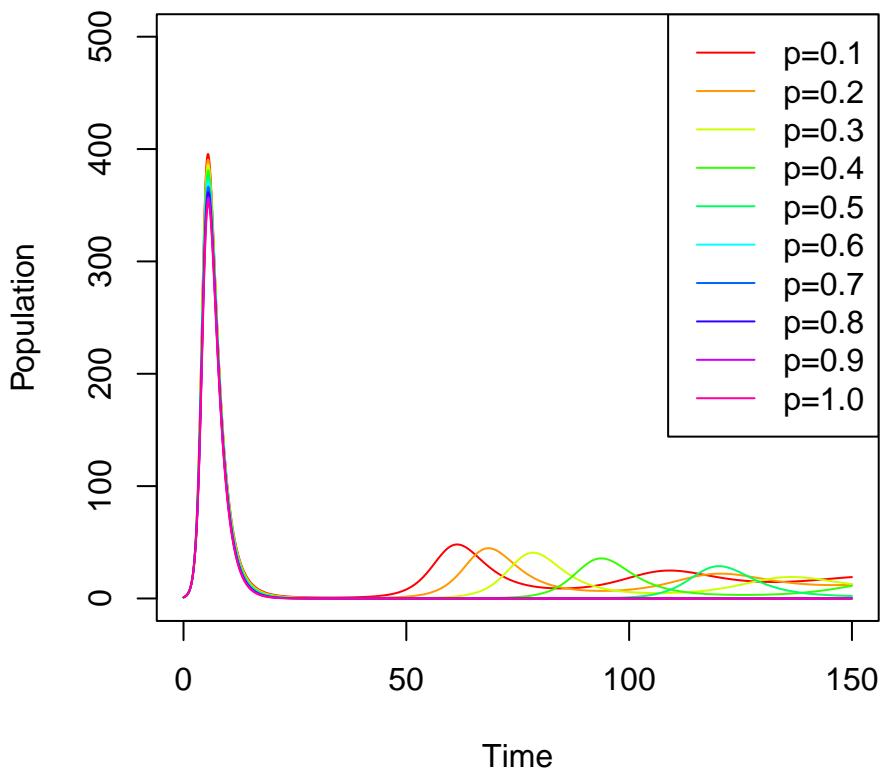
  det.sol <- lsoda(SIR.init,SIR.t,SIR.dyn,SIR.par)

  # plot I curve only

  lines(SIR.t,det.sol[,3],col=col_vals[i])
}

legend('topright',c('p=0.1',
                    'p=0.2',
                    'p=0.3',
                    'p=0.4',
                    'p=0.5',
                    'p=0.6',
                    'p=0.7',
                    'p=0.8',
                    'p=0.9',
                    'p=1.0'),col=col_vals,lty=1)

```



[Return to task on P100](#)

Answer 59

We can see that for $p < p_{vac}$, we have a second outbreak following the initial first outbreak. This does not occur if $p > p_{vac}$. We still get an initial epidemic in all cases because we start with an entirely susceptible population. Since we only vaccinate individuals as they are born, it takes some time for the total proportion of population vaccinated to be high enough to ensure herd immunity.

[Return to task on P100](#)

Solution 6o

```

# Values of vaccination coverage to try
p_vals = seq(0.1,1.0,0.1)
# Color palette for plotting
col_vals = rainbow(length(p_vals))

# Make an empty plot
plot(0,xlim=c(0,150),ylim=c(0,1000),pch=' ',xlab='Time',ylab='Population')

# Add herd immunity threshold

abline(h=(1-26/35)*1000,col='black',lwd=2)

# Initial conditions and time points same for each solution
SIR.init<-c(S=999,I=1,R=0)
SIR.t<-seq(0,150,0.1)

for(i in 1:length(p_vals)) {
  # Change vaccination coverage
  SIR.par<-c(beta=2,gamma=0.5,mu=1/70.0,p=p_vals[i])

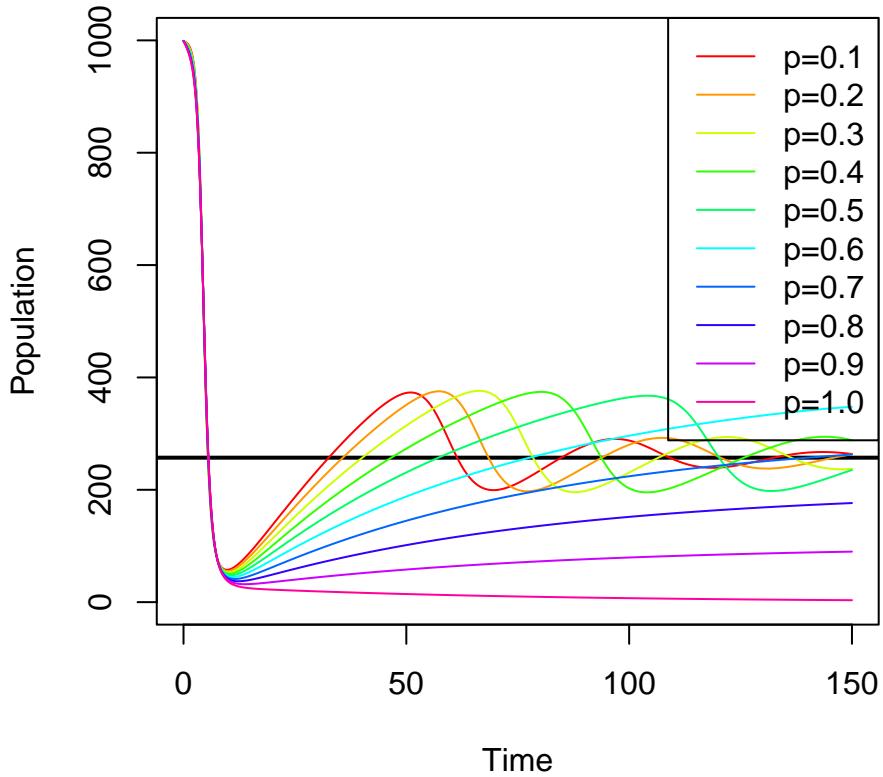
  det.sol <- lsoda(SIR.init,SIR.t,SIR.dyn,SIR.par)

  # plot S curve only

  lines(SIR.t,det.sol[,2],col=col_vals[i])
}

legend('topright',c('p=0.1',
                    'p=0.2',
                    'p=0.3',
                    'p=0.4',
                    'p=0.5',
                    'p=0.6',
                    'p=0.7',
                    'p=0.8',
                    'p=0.9',
                    'p=1.0'),col=col_vals,lty=1)

```



In all cases except $p = 1$, we tend towards the steady state at $\frac{1}{R_0}$ (shown in black). If $p < p_{vac}$, the proportion of susceptibles increases after the initial peak and overshoots $\frac{1}{R_0}$ before fluctuating and settling to the steady state; otherwise the number of susceptibles slowly increases to the steady state.

[Return to task on P100](#)

Solution 61

First consider one-off campaign on its own without vaccination at birth.

```

# Values of vaccination coverage to try
p_vals = seq(0.1,1.0,0.1)
# Color palette for plotting
col_vals = rainbow(length(p_vals))
# Vaccination at birth now 0
SIR.par<-c(beta=2,gamma=0.5,mu=1/70.0,p=0.0)

# Make an empty plot
plot(0,xlim=c(0,150),ylim=c(0,500),pch=' ',xlab='Time',ylab='Population')

# Time points same for each solution

SIR.t<-seq(0,150,0.1)

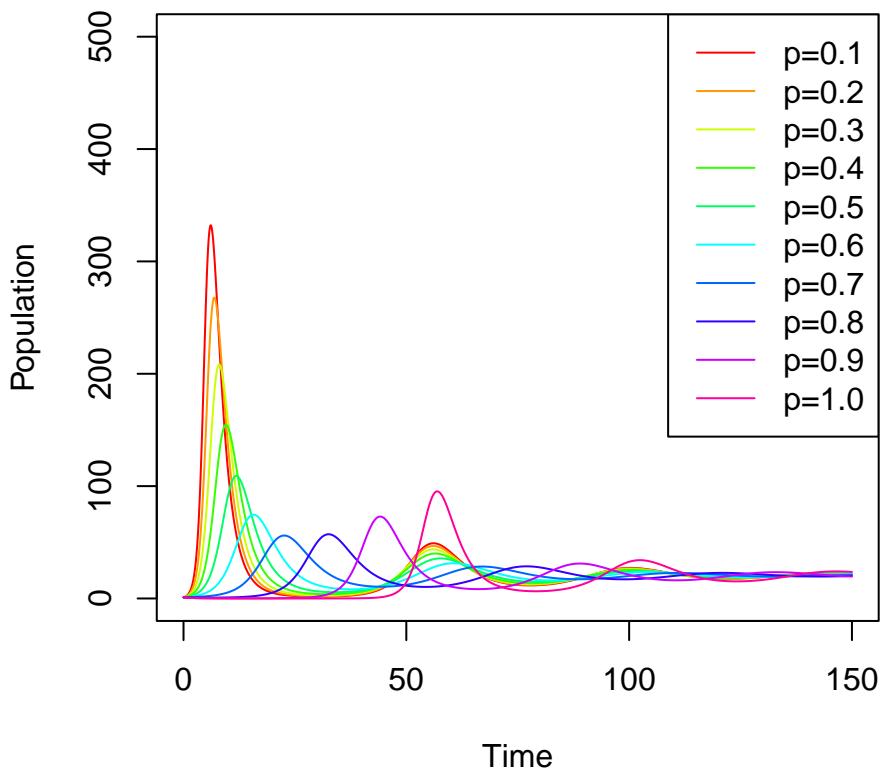
for(i in 1:length(p_vals)) {
  # Now model one-off vaccination campaign at start by modifying the initial conditions
  SIR.init<-c(S=(999)*(1-p_vals[i]),I=1,R=(999)*(p_vals[i]))

  det.sol <- lsoda(SIR.init,SIR.t,SIR.dyn,SIR.par)

  # plot I curve only
  lines(SIR.t,det.sol[,3],col=col_vals[i])
}

legend('topright',c('p=0.1',
                    'p=0.2',
                    'p=0.3',
                    'p=0.4',
                    'p=0.5',
                    'p=0.6',
                    'p=0.7',
                    'p=0.8',
                    'p=0.9',
                    'p=1.0'),col=col_vals,lty=1)

```



A single one off campaign can delay - but not prevent an epidemic. Even for $p > 0.75$, there will still be a smaller outbreak once enough births have accumulated to bring the number of susceptibles above the herd immunity threshold.

If we now consider a one-off campaign in addition to vaccination at birth at 75%:

```

# Values of vaccination coverage to try
p_vals = seq(0.1,1.0,0.1)
# Color palette for plotting
col_vals = rainbow(length(p_vals))
# Vaccination at birth now 0.75
SIR.par<-c(beta=2,gamma=0.5,mu=1/70.0,p=0.75)

# Make an empty plot
plot(0,xlim=c(0,150),ylim=c(0,500),pch=' ',xlab='Time',ylab='Population')

# Time points same for each solution

SIR.t<-seq(0,150,0.1)

for(i in 1:length(p_vals)) {
  # Now model one-off vaccination campaign at start by modifying the initial conditions
  SIR.init<-c(S=(999)*(1-p_vals[i]),I=1,R=(999)*(p_vals[i]))

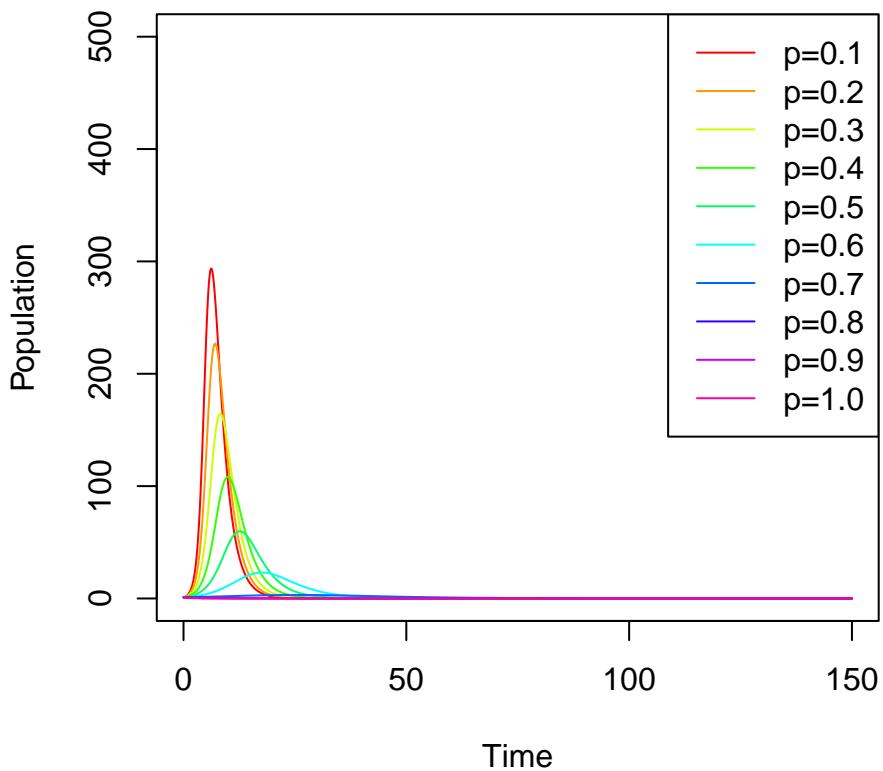
  det.sol <- lsoda(SIR.init,SIR.t,SIR.dyn,SIR.par)

  # plot I curve only

  lines(SIR.t,det.sol[,3],col=col_vals[i])
}

legend('topright',c('p=0.1',
                   'p=0.2',
                   'p=0.3',
                   'p=0.4',
                   'p=0.5',
                   'p=0.6',
                   'p=0.7',
                   'p=0.8',
                   'p=0.9',
                   'p=1.0'),col=col_vals,lty=1)

```



Now we can see that increasing p for an initial campaign not only decreases the size of the initial outbreak, but for $p > p_{vac}$ for the initial campaign, there is now no outbreak.

[Return to task on P100](#)

Answer 62

This suggests that birth programmes used in isolation are insufficient to prevent outbreaks; we also need a sufficiently large proportion of the population immune to begin with.

[Return to task on P100](#)

Solution 63

$$K = \frac{1}{\gamma} \beta = \begin{pmatrix} 4 & 0.4 \\ 0.4 & 2 \end{pmatrix}$$

[Return to task on P103](#)

Solution 64

Using the formula for R_0 from the lectures:

$$R_0 = \frac{4+2+\sqrt{(4+2)^2-4(4\times 2-0.4^2)}}{2} \simeq 4.08$$

The critical vaccination threshold is

```
# Next Generation Matrix and R0

ngm = (4.0)*matrix(c(1,0.1,0.1,0.5),byrow=T,nrow=2)

R0 = max(eigen(ngm)$values)

p_critical = 1 - 1/R0
```

[Return to task on P103](#)

Solution 65

```
# Values of vaccination coverage to try
p_vals = seq(0.1,1.0,0.1)
# Color palette for plotting
col_vals = rainbow(length(p_vals))

# Make an empty plot
plot(0,xlim=c(0,50),ylim=c(0,25),pch=' ',xlab='Time',ylab='Population')

for(i in 1:length(p_vals)) {
  # Now model one-off vaccination campaign at start by modifying the initial conditions
  init <- c(49*(1-p_vals[i]),50*(1-p_vals[i]),1,0,49*p_vals[i],50*p_vals[i])

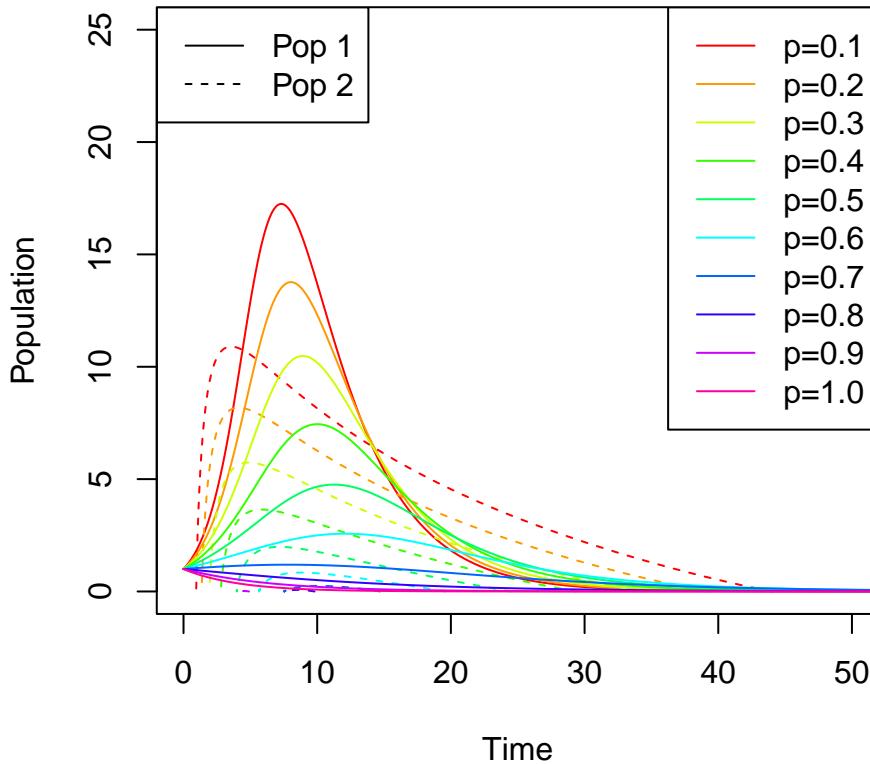
  TwoPop.sol <- lsoda(init,TwoPop.t,TwoPopSIR,TwoPop.par)

  # plot I curve only

  lines(TwoPop.sol[,1],TwoPop.sol[,4],col=col_vals[i],lty=1)
  lines(TwoPop.sol[,2],TwoPop.sol[,5],col=col_vals[i],lty=2)
}

legend('topright',c('p=0.1',
                    'p=0.2',
                    'p=0.3',
                    'p=0.4',
                    'p=0.5',
                    'p=0.6',
                    'p=0.7',
                    'p=0.8',
                    'p=0.9',
                    'p=1.0'),col=col_vals,lty=1)

legend('topleft',c('Pop 1','Pop 2'),col='black',lty=c(1,2))
```



[Return to task on P103](#)

Solution 66

```
# Critical proportion (p2) that needs to be vaccinated given a value of p1
# Using equation from lectures

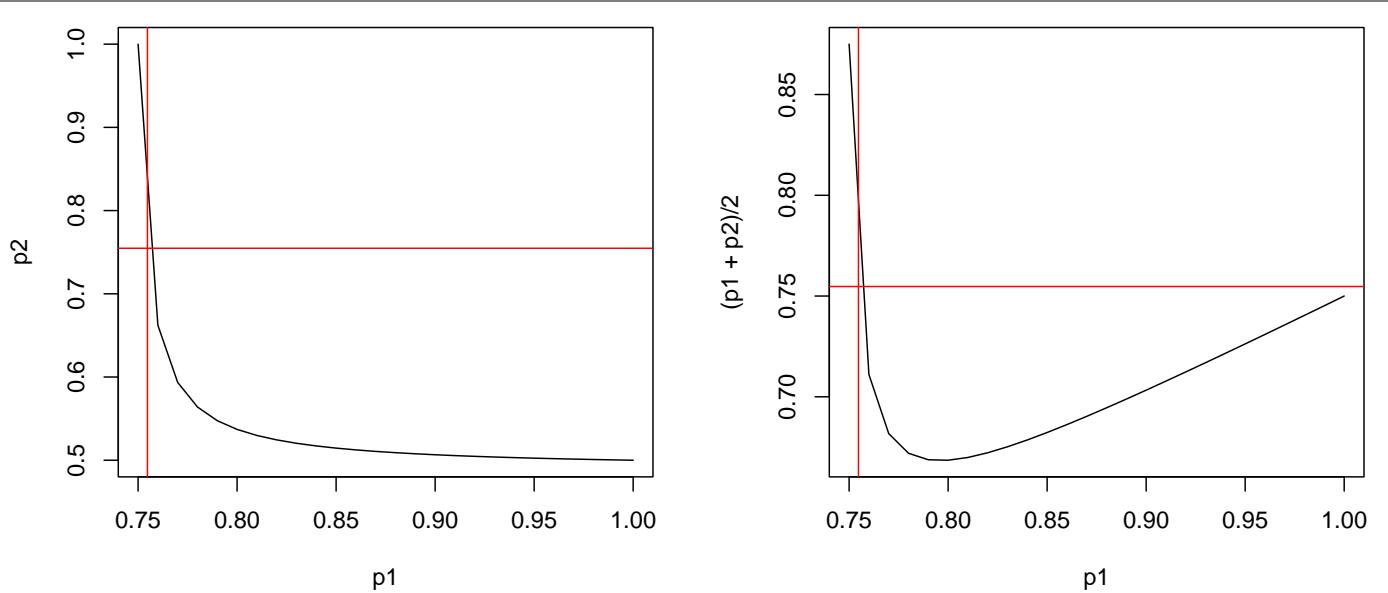
a = ngm[1,1];b = ngm[1,2]; c = ngm[2,1]; d = ngm[2,2]

# p1 must be greater than 1 - 1/a
p1 = seq(1-1/a,1,0.01)

p2 = 1 - ((1 - a*(1-p1))/(d - (a*d - b*c)*(1-p1)))

par(mfrow=c(1,2))

plot(p1,p2,type='l')
abline(v=p_critical,col='red')
abline(h=p_critical,col='red')
plot(p1,(p1+p2)/2,type='l')
abline(v=p_critical,col='red')
abline(h=p_critical,col='red')
par(mfrow=c(1,1))
```



We can read off the minimum possible value of p off our graph. This gives $p \simeq 0.668$, $p_1 \simeq 0.796$ and $p_2 \simeq 0.541$. Overall, this is a lower level of vaccination than random vaccination, so may be more efficient. The optimal vaccination scenario below, note that although the number of infectives in the second population initially increase it never crosses 1.

```

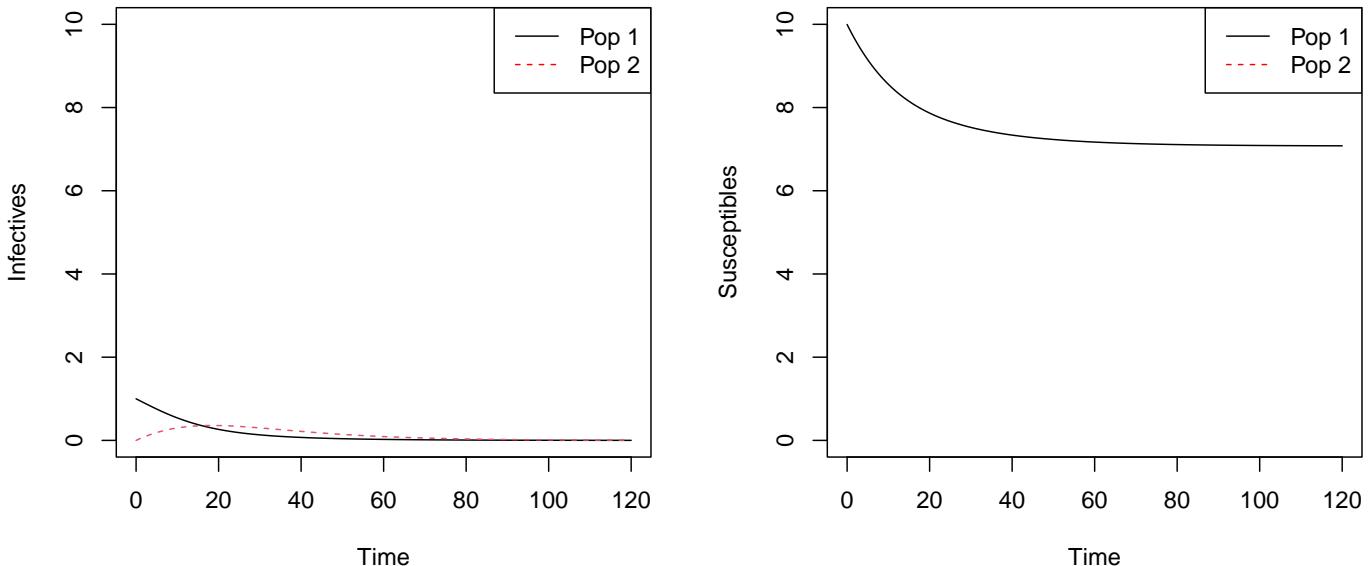
p1=0.796
p2=0.541
Optimal.init<- c(49*(1-p1),50*(1-p2),1,0,49*p1,50*p2)

TwoPop.t<-seq(0,120,0.1)
# Note, at optimal coverage I[2] initially increases, but never crosses 1...

TwoPop.sol <- lsoda(Optimal.init,TwoPop.t,TwoPopSIR,TwoPop.par)

par(mfrow=c(1,2))
matplot(TwoPop.sol[,1],TwoPop.sol[,c(4,5)],type='l',xlab='Time',ylab='Infectives',ylim=c(0,10))
legend('topright',c('Pop 1','Pop 2'),col=c('black','red'),lty=c(1,2))
matplot(TwoPop.sol[,1],TwoPop.sol[,c(2,3)],type='l',xlab='Time',ylab='Susceptibles',ylim=c(0,10))
legend('topright',c('Pop 1','Pop 2'),col=c('black','red'),lty=c(1,2))
par(mfrow=c(1,1))

```

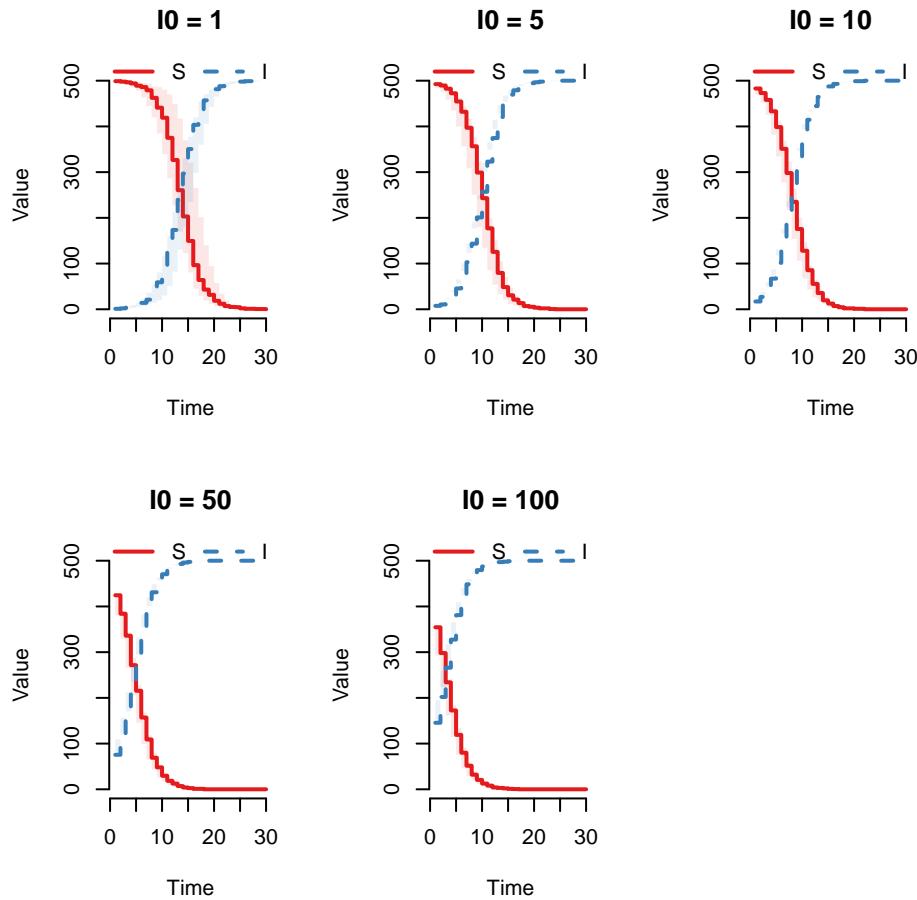


[Return to task on P104](#)

Solution 67

As the initial number of infectives increases the variability between replicates decreases.

```
ninf <- c(1, 5, 10, 50, 100)
par(mfrow = c(2, 3))
for(i in ninf) {
  SImodel <- create.StocSI(500, i, 0.5, 30, 20)
  out <- run(model = SImodel)
  plot(out, main = paste0("I0 = ", i))
}
par(mfrow = c(1, 1))
```



[Return to task on P112](#)

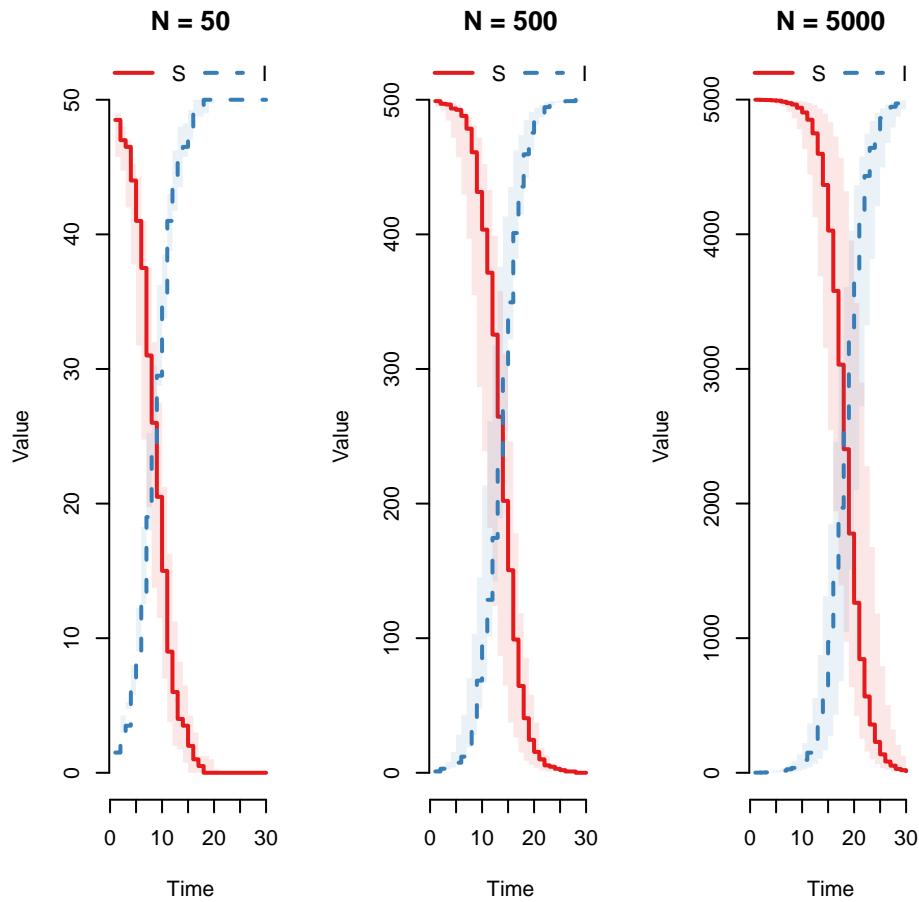
Solution 68

In contrast to varying the initial number of infectives, the host population size has little to no impact on the variability between replicates of the *SI* model. The relevant “large population” limit for the *SI* model is the number of infectives not the size of the host population.

```

npop <- c(50, 500, 5000)
par(mfrow = c(1, 3))
for(i in npop) {
  SImodel <- create.StocSI(i, 1, 0.5, 30, 20)
  out <- run(model = SImodel)
  plot(out, main = paste0("N = ", i))
}
par(mfrow = c(1, 1))

```



[Return to task on P113](#)

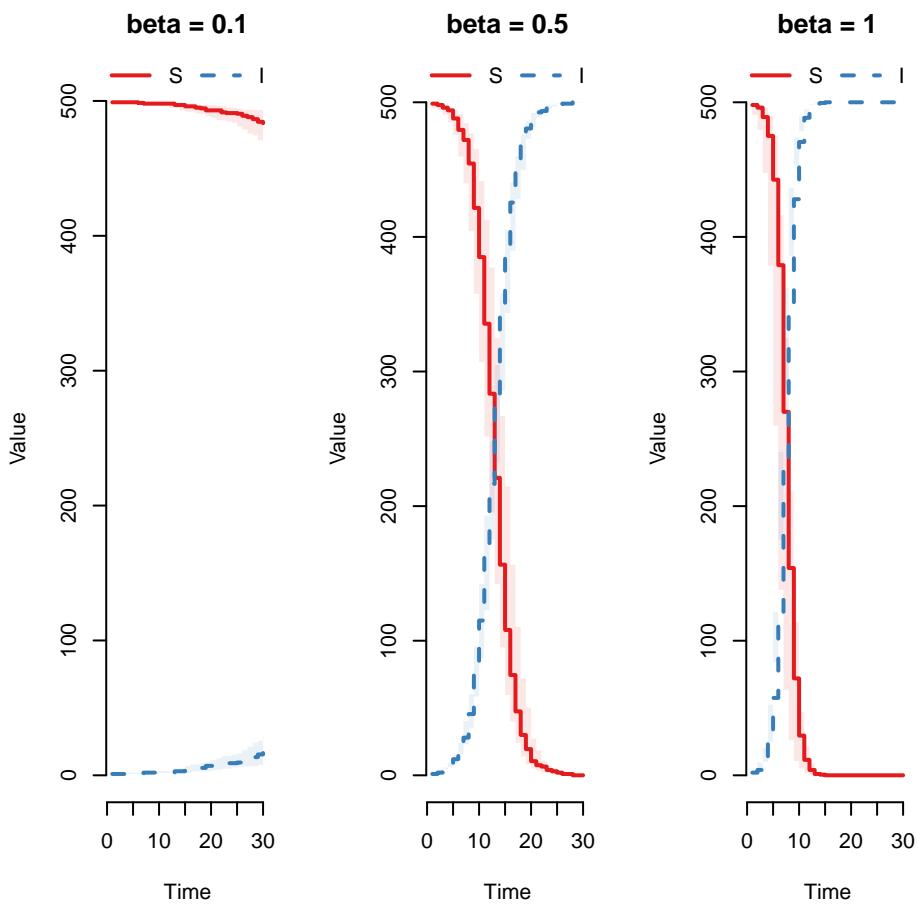
Solution 69

As discussed in lectures, increasing the transmission rate reduces the impact of stochasticity and the variability between replicates.

```

beta <- c(0.1, 0.5, 1)
par(mfrow = c(1, 3))
for(i in beta) {
  SImodel <- create.StocSI(500, 1, i, 30, 20)
  out <- run(model = SImodel)
  plot(out, main = paste0("beta = ", i))
}
par(mfrow = c(1, 1))

```



[Return to task on P113](#)

Solution 72

```
create.StocSIR <- function(N, I0, beta, mu, f_time, reps = 1) {
  initial_state <- data.frame(S = rep(N-I0, reps), I = rep(I0, reps), R = rep(0, reps))
  compartments <- c("S", "I", "R")
  transitions <- c("S -> beta*S*I/(S+I+R) -> I", "I -> mu*I -> R")
  tspan <- seq(from = 1, to = f_time, by = 1)

  model <- mparse(transitions = transitions,
                    compartments = compartments,
                    gdata = c(beta = beta, mu = mu),
                    u0 = initial_state,
                    tspan = tspan)

  return(model)
}
SIRmodel <- create.StocSIR(500, 1, 1, 0.25, 50)
out <- run(model = SIRmodel)
plot(out)
```

[Return to task on P118](#)

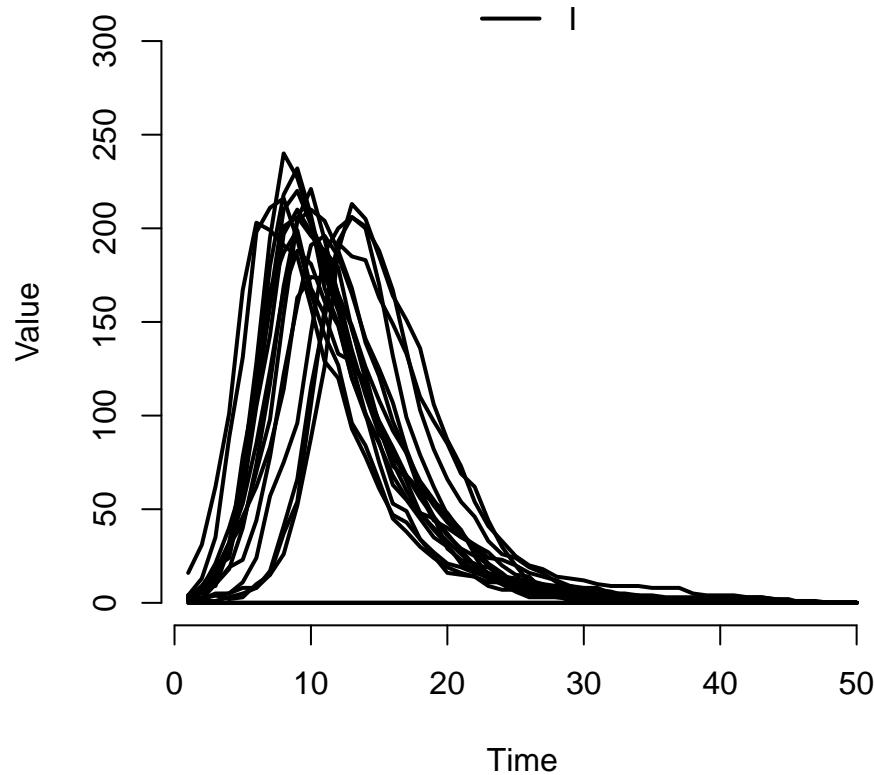
Answer 73

For some replicates the initial infected individual will recover before transmitting, the disease will go extinct and there will be no epidemic.

[Return to task on P119](#)

Solution 74

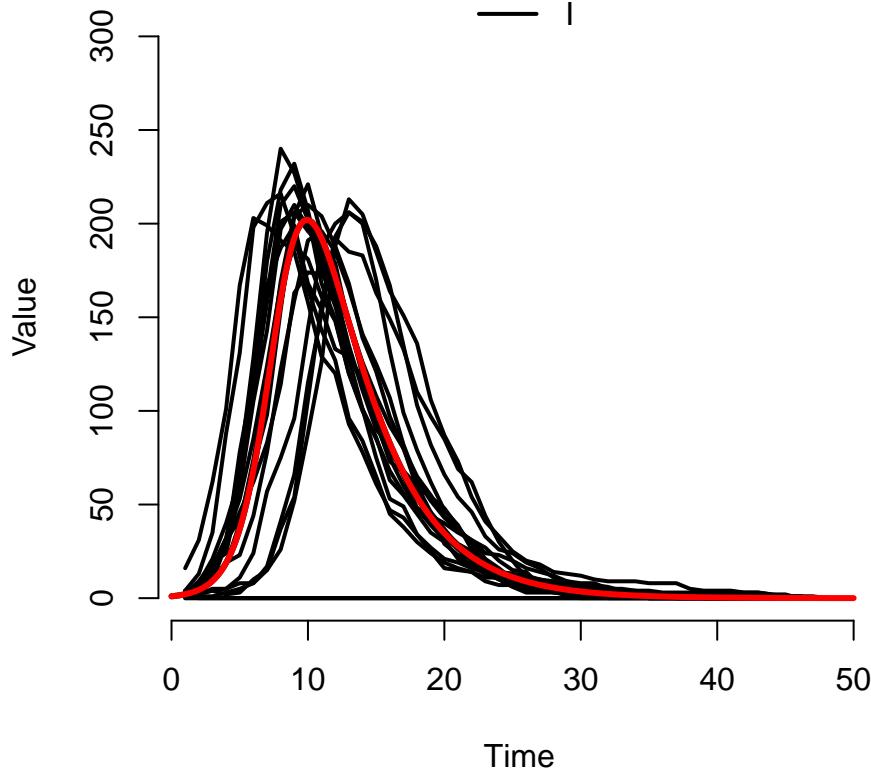
```
SIRmodel <- create.StocSIR(500, 1, 1, 0.25, 50, 20)
out <- run(model = SIRmodel)
plot(out , 'I', range=FALSE , type = 'l', ylim = c(0, 300))
```



[Return to task on P119](#)

Solution 75

```
source('Materials/Epidemic_models/Functions/Det_SIR.R')
det.sol <- DetSIR.dyn(500, 1, 1, 0.25, 50)
det.t <- det.sol[,1]
det.I <- det.sol[,3]
lines(det.t, det.I, lwd = 3, col = 'red')
```



[Return to task on P119](#)

Solution 76

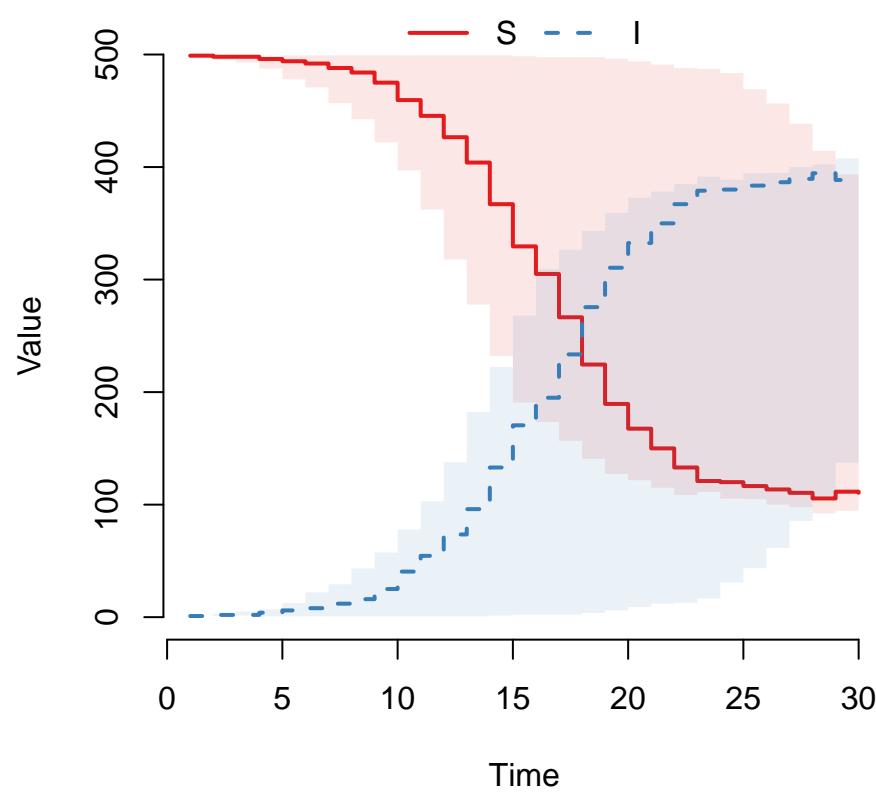
```

create.StocSIS <- function(N, I0, alpha, beta, f_time, reps = 1) {
  initial_state <- data.frame(S = rep(N-I0, reps), I = rep(I0, reps))
  compartments <- c("S", "I")
  transitions <- c("S -> beta*S*I/(S+I) -> I", "I -> alpha*I -> S")
  tspan <- seq(from = 1, to = f_time, by = 1)

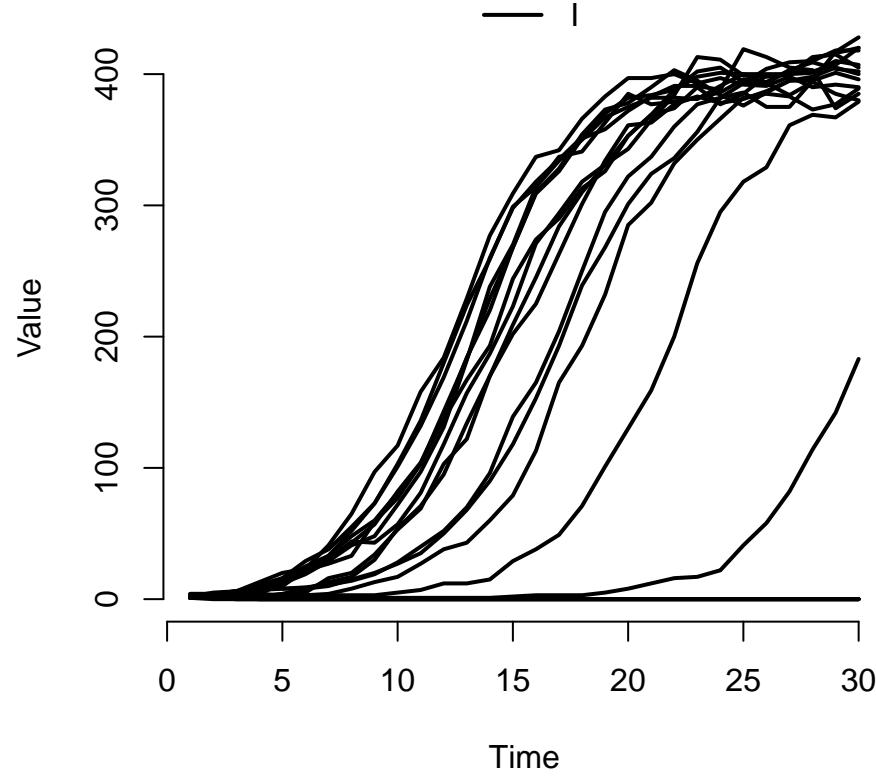
  model <- mparse(transitions = transitions,
                    compartments = compartments,
                    gdata = c(alpha = alpha, beta = beta),
                    u0 = initial_state,
                    tspan = tspan)

  return(model)
}
SISmodel <- create.StocSIS(500, 1, 0.1, 0.5, 30, 20)
out <- run(model = SISmodel)
plot(out)

```



```
plot(out , 'I', range = FALSE , type = 'l')
```



[Return to task on P120](#)

Solution 77

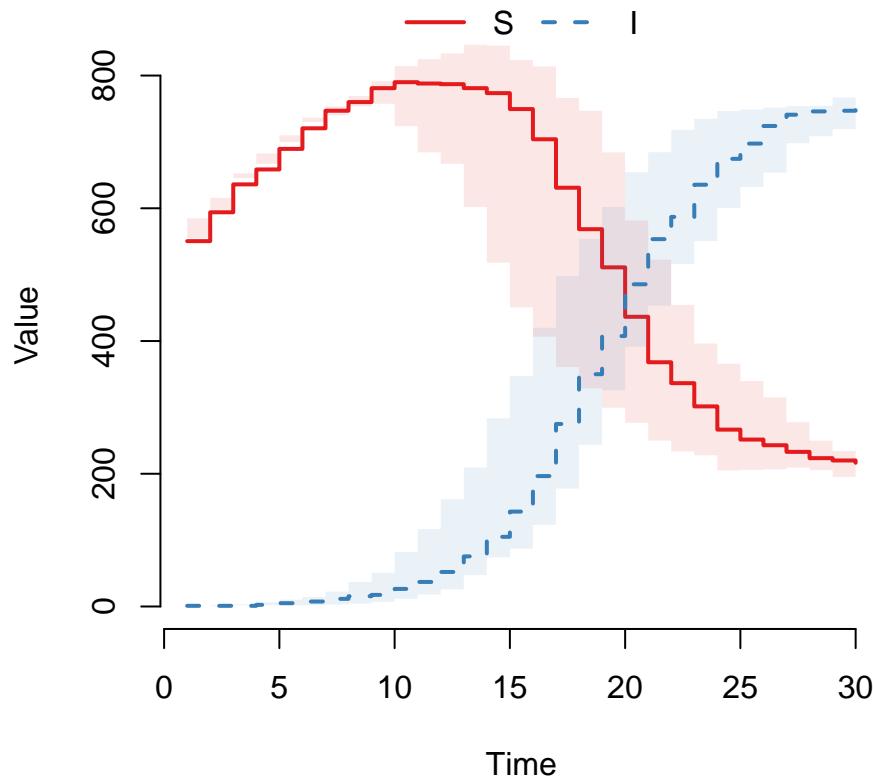
```

create.StocSI_demo <- function(N, I0, mu, beta, lambda, f_time, reps = 1) {
  initial_state <- data.frame(S = rep(N-I0, reps), I = rep(I0, reps))
  compartments <- c("S", "I")
  transitions <- c(
    "@ -> lambda -> S",
    "S -> beta*S*I/(S+I) -> I",
    "S -> mu*S -> @",
    "I -> mu*I -> @"
  )
  tspan <- seq(from = 1, to = f_time, by = 1)

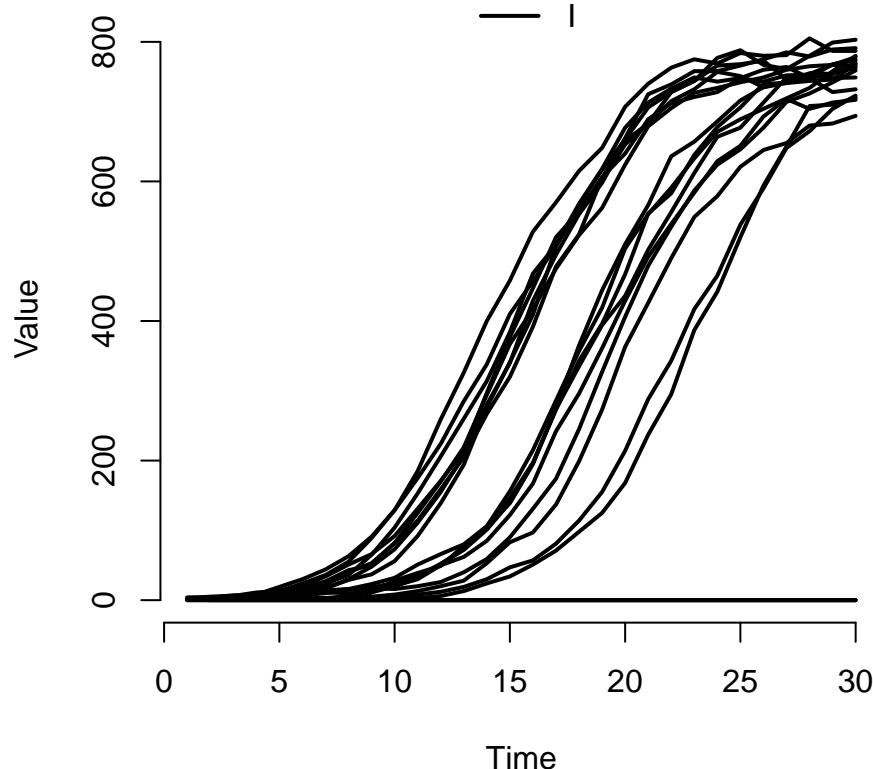
  model <- mparse(transitions = transitions,
                    compartments = compartments,
                    gdata = c(mu = mu, beta = beta, lambda = lambda),
                    u0 = initial_state,
                    tspan = tspan)

  return(model)
}
SI_demo_model <- create.StocSI_demo(500, 1, 0.1, 0.5, 100, 30, 20)
out <- run(model = SI_demo_model)
plot(out)

```



```
plot(out, 'I', range = FALSE, type = 'l')
```



[Return to task on P121](#)

Solution 78

Here we will extract the number of removals at time $t = 50$ as an approximation of the final epidemic size.

```
## check for a single simulation
SIRmodel <- create.StocSIR(500, 1, 1, 0.25, 50)
out <- run(model = SIRmodel)

## extract states
u <- trajectory(out)

## find approximate final epidemic size
finalsize <- u$R[u$time == 50]
finalsize

## [1] 491
```

[Return to task on P121](#)

Solution 79

```
## run 100 replicates
SIRmodel <- create.StocSIR(500, 1, 1, 0.25, 50, 100)
out <- run(model = SIRmodel)

## extract states
u <- trajectory(out)

## find approximate final epidemic size for each replicate
finalsize <- rep(NA, 100)
for(i in 1:100) {
  finalsize[i] <- u$R[u$node == i & u$time == 50]
}
hist(finalsize, breaks = 500)
```

This has a bimodal distribution because as before, with a small number of initial infectious individuals there is a non-zero probability that these early infections will recover before they infect anyone else. If an epidemic does take off then the final sizes tend to cluster around the deterministic final size ($\approx N = 500$ here).

[Return to task on P122](#)

Solution 80

$$R_0 = \frac{\beta}{\gamma} = \frac{1}{0.25} = 4.$$

From the lecture notes we have that the probability of early extinction is $P(E) \approx \frac{1}{R_0} = 0.25$. We can produce an approximation of the early extinction probabilities by counting the proportion of realisations that die out early. Based on the final size graph above, given the two very distinct modes, if we define early extinction as being any realisation with a final size < 250 say, then we can generate an estimate for $P(E)$ as:

```
PE <- sum(finalsize < 250) / length(finalsize)
PE
```

```
## [1] 0.2
```

We can see that this closely aligns to our theoretical approximation.

[Return to task on P122](#)

Solution 81

```

## set vector of R0 values
R0 <- c(20, 5, 2, 1.2, 1.0, 0.9, 0.5)

## set mu
mu <- 0.25

## set plot layout
par(mfrow = c(3, 3))

## loop over R0 values
for(i in 1:length(R0)) {

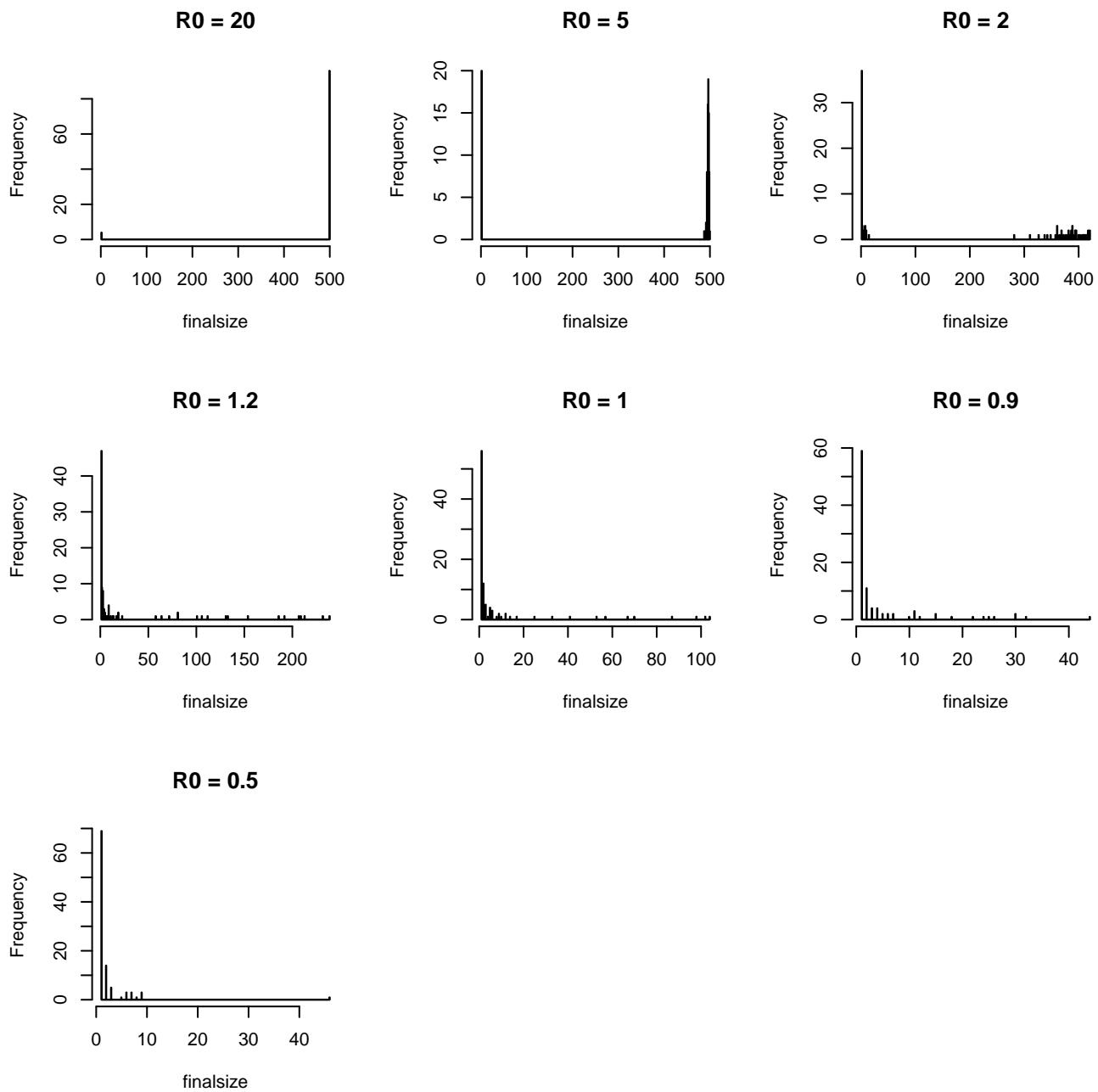
    ## calculate beta for fixed mu and R0
    beta <- R0[i] * mu

    ## set up and run model
    SIRmodel <- create.StocSIR(500, 1, beta, mu, 50, 100)
    out <- run(model = SIRmodel)

    ## extract states
    u <- trajectory(out)

    ## find final epidemic size for each replicate
    finalsize <- rep(NA, 100)
    for(j in 1:100) {
        finalsize[j] <- u$R[u$node == j & u$time == 50]
    }
    hist(finalsize, breaks = 500, main = paste0("R0 = ", R0[i]))
}
par(mfrow = c(1, 1))

```



For intermediate values of R_0 the two modes will merge together.

[Return to task on P123](#)

Answer 82

We should monitor the first time at which the number of infectious individuals, I , equals 0.

[Return to task on P123](#)

Solution 83

To generate the first figure, consider:

```

set.seed(126)
## run 100 replicates
SIRmodel <- create.StocSIR(500, 1, 1, 0.25, 50, 100)
out <- run(model = SIRmodel)

## extract states
u <- trajectory(out)

## find extinction time for each replicate
finaltime <- rep(NA, 100)
for(i in 1:100) {
  ## note that this will return an NA if the epidemic hasn't ended
  finaltime[i] <- u$time[u$node == i & u$I == 0][1]
}
hist(finaltime)

```

To then explore how this distribution changes for different values of R_0 , consider:

```

## set vector of R0 values
R0 <- c(20, 5, 2, 1.2, 1.0, 0.9, 0.5)

## set mu
mu <- 0.25

## set plot layout
par(mfrow = c(3, 3))

## loop over R0 values
for(i in 1:length(R0)) {

  ## calculate beta for fixed mu and R0
  beta <- R0[i] * mu

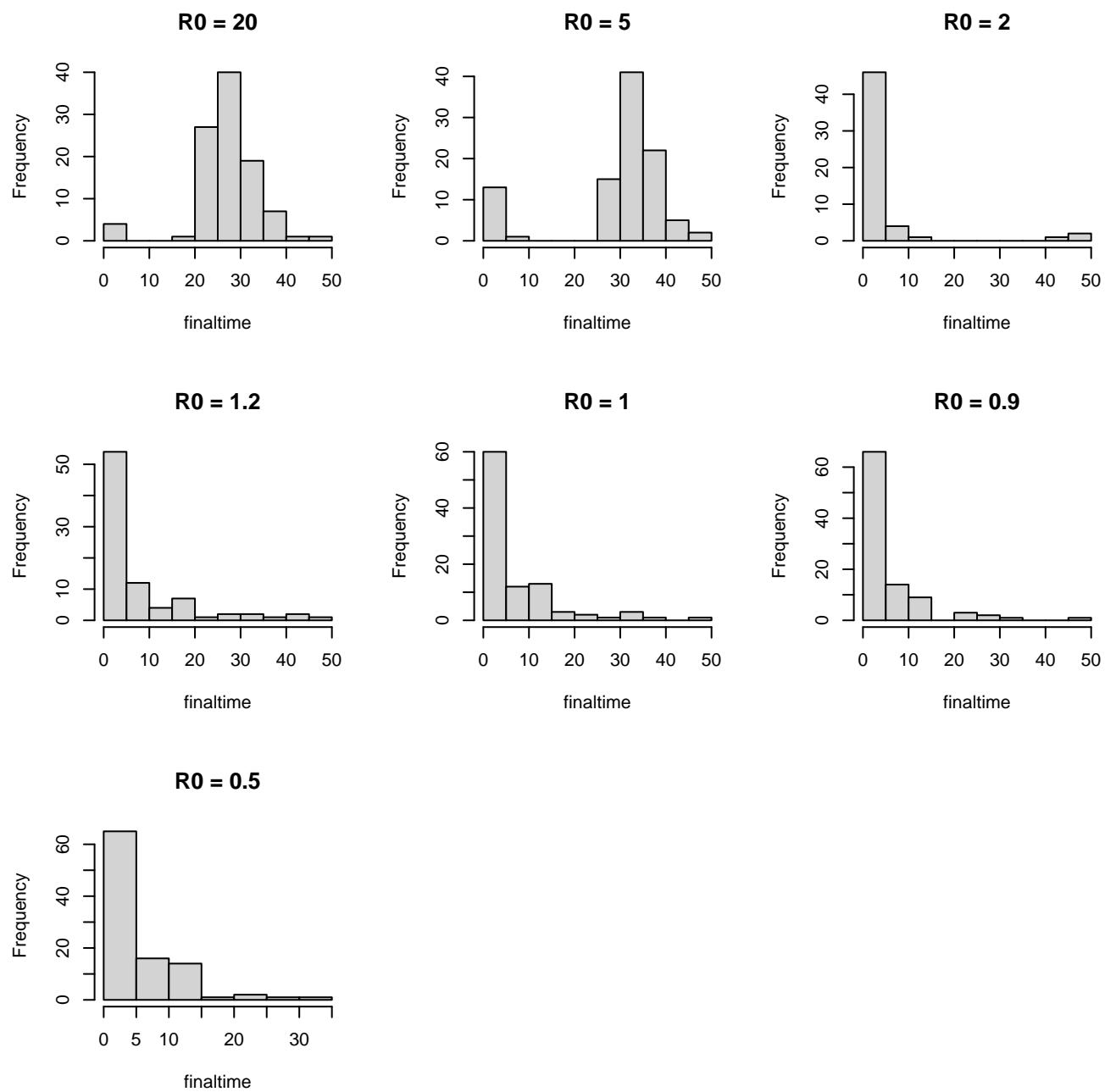
  ## set up and run model
  SIRmodel <- create.StocSIR(500, 1, beta, mu, 50, 100)
  out <- run(model = SIRmodel)

  ## extract states
  u <- trajectory(out)

  ## find extinction time for each replicate
  finaltime <- rep(NA, 100)
  for(j in 1:100) {
    ## note that this will return an NA if the epidemic hasn't ended
    finaltime[j] <- u$time[u$node == j & u$I == 0][1]
  }
  hist(finaltime, main = paste0("R0 = ", R0[i]))
}

par(mfrow = c(1, 1))

```



As R_0 gets closer to one, the extinction time distribution loses the upper mode and moves far closer to zero.

[Return to task on P124](#)

Solution 84

```
rubella$p_s <- 1 - rubella$Pos / (rubella$Neg + rubella$Pos)
```

[Return to task on P128](#)

Solution 85

$$\begin{aligned}\log(p_1) &= \log(p_0 e^{-\lambda \Delta a}) \\ \Rightarrow \log(p_1) &= \log(p_0) - \lambda \Delta a \\ \Rightarrow \lambda \Delta a &= \log(p_0) - \log(p_1) \\ \Rightarrow \lambda &= \frac{1}{\Delta a} [\log(p_0) - \log(p_1)],\end{aligned}$$

which can also be written as:

$$\lambda = \frac{1}{\Delta a} \log\left(\frac{p_0}{p_1}\right).$$

[Return to task on P129](#)

Solution 86

If $\Delta a = 1$ for all age-cohorts, then the equation for λ_i can be estimated within each age-cohort as:

$$\lambda_i = \log(p_{i-1}) - \log(p_i)$$

for $i = 1, \dots, N_A$ where $p_0 = 1$. This can be implemented in R using e.g.

```
rubella$lambda <- log(c(1, rubella$p_s[1:(nrow(rubella) - 1)])) -
  log(rubella$p_s)
```

[Return to task on P129](#)

Answer 87

Here some of the estimates of λ are *negative*, which doesn't make sense biologically. Remember, we are crudely estimating λ multiple times using only information in each cohort, whilst putting no constraints on the range of values that the λ terms can take. We are also ignoring the fact that the susceptible proportions in each age-class are themselves estimates, and are thus measured with uncertainty. This is not a robust approach for statistical inference!

[Return to task on P129](#)

Solution 88

There is a suggestion from Figure 9.3 that infants, as well as adults, have a lower rate of infection than school age children. Such a "core-group" model will provide a closer fit to the serological age-profile.

```

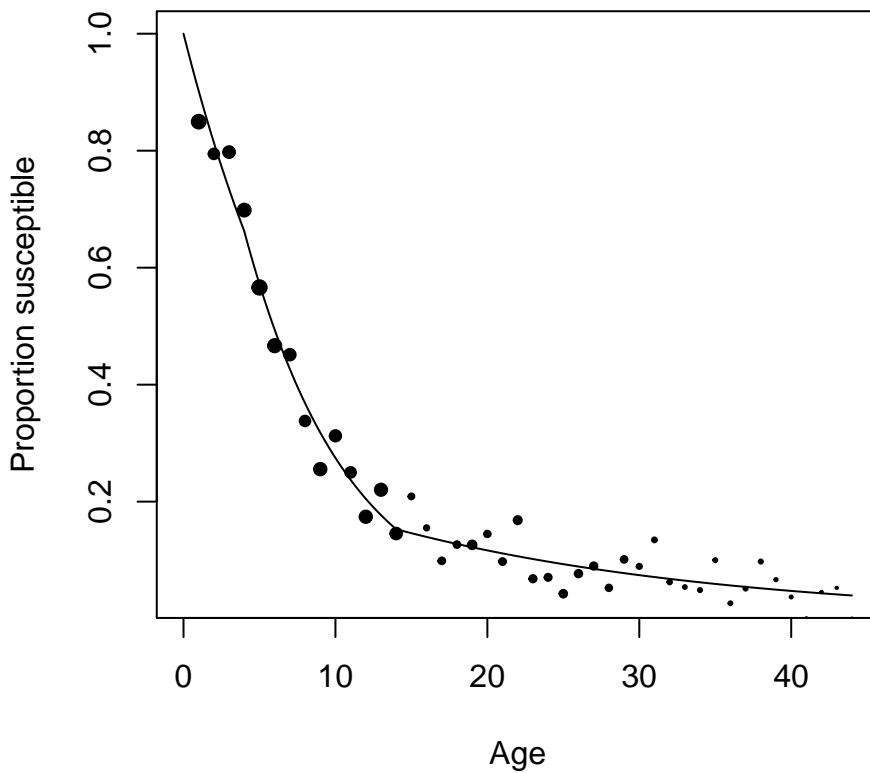
## set up FOI mappings
three_groups <- rep(NA, max(rubella$Age))
three_groups[1:max(rubella$Age) < 5] <- 1
three_groups[1:max(rubella$Age) >= 5 & 1:max(rubella$Age) < 15] <- 2
three_groups[1:max(rubella$Age) >= 15] <- 3

## fit model
threegroup_mle <- optim(
  c(0.07, 0.07, 0.07),
  FOI_loglik,
  foi_key = three_groups,
  sero_data = rubella,
  control = list(fnscale = -1)
)
threegroup_mle

## $par
## [1] 0.10299635 0.14664689 0.04503626
##
## $value
## [1] -105.7169
##
## $counts
## function gradient
##       88      NA
##
## $convergence
## [1] 0
##
## $message
## NULL

## plot model fits
plot(
  ages,
  piecewise_s(ages, threegroup_mle$par[three_groups]),
  type = 'l',
  xlab = "Age",
  ylab = "Proportion susceptible"
)
points(rubella$Age, rubella$p_s, pch = 19, cex = size^0.75)

```



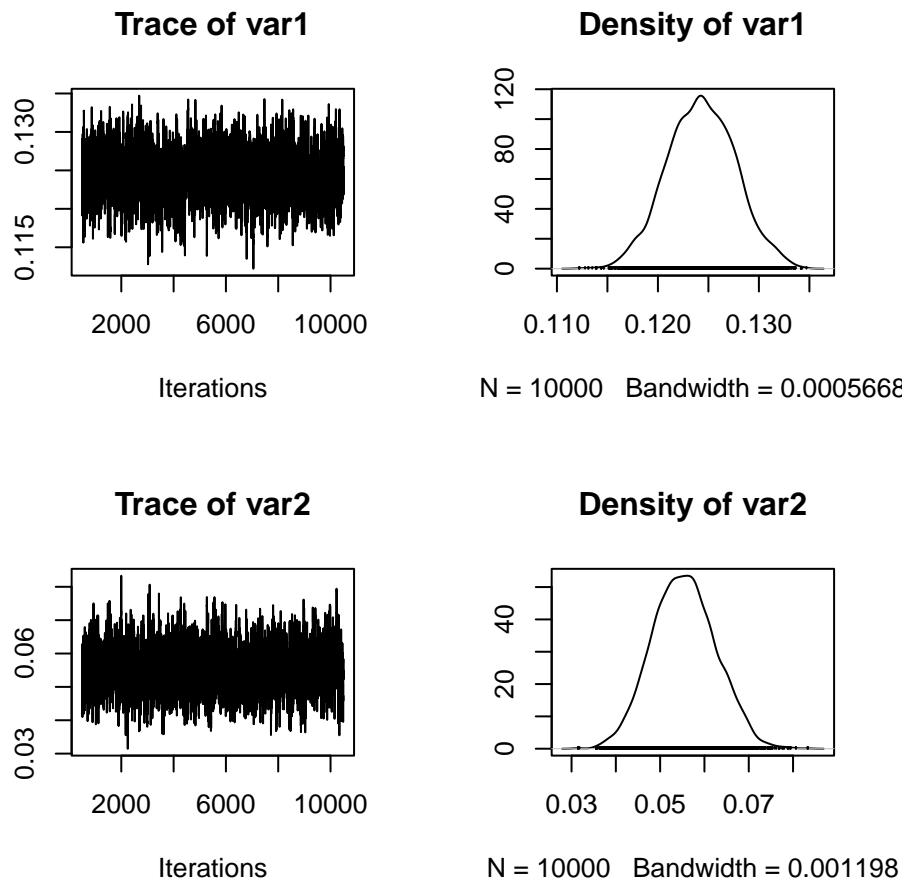
The log-likelihood of -106 compared to -111 for the two group model, provides evidence that the three-group model better explains the data even allowing for any penalties we might chose to impose due to the additional parameter (e.g. AIC).

[Return to task on P133](#)

Solution 89

To check the chains we can simply plot the `two_group_post` object:

```
plot(twogroup_post)
```



```
summary(twogroup_post)
```

```
##
## Iterations = 501:10500
## Thinning interval = 1
## Number of chains = 1
## Sample size per chain = 10000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##          Mean        SD   Naive SE Time-series SE
## [1,] 0.12432 0.003374 3.374e-05     0.0001069
## [2,] 0.05539 0.007129 7.129e-05     0.0002193
##
## 2. Quantiles for each variable:
##
##      2.5%    25%    50%    75%   97.5%
## var1 0.11765 0.1220 0.12432 0.12667 0.13112
## var2 0.04191 0.0504 0.05521 0.06025 0.06946
```

Here the chains look well mixed and the algorithm seems to have converged (ideally we would want to run multiple chains). The 95% credible intervals for the two force of infection parameters are non-overlapping with a risk of infection for school age-children almost twice that of adults.

[Return to task on P134](#)

Solution 90

For the ‘infant’ matrix:

$$k_1 = \frac{\lambda_1}{\psi_1} \quad \text{and} \quad k_2 = \frac{\lambda_2}{\psi_1}.$$

For the ‘diagonal’ matrix:

$$k_1 = \frac{\lambda_1}{\psi_1} \quad \text{and} \quad k_2 = \frac{\lambda_2}{\psi_2}.$$

[Return to task on P136](#)

Solution 91

```
## set up mortality rate and infectious period
u <- 20 / 1000
TI <- 5 / 365

## calculate psi
psi <- Psi(twogroup_mle$par, age_widths = c(15, 30), u, TI)

## proportion of population in the two age groups (exponential
## age-distribution)
n <- c((1 - exp(-15 * u)), exp(-15 * u))

## infant mixing MLE
k1 <- twogroup_mle$par[1] / psi[1]
k2 <- twogroup_mle$par[2] / psi[1]
max(TI * eigen(matrix(c(k1, k2, 0, 0), 2, 2) * n)$values)

## [1] 1.905538

## diagonal mixing MLE
k1 <- twogroup_mle$par[1] / psi[1]
k2 <- twogroup_mle$par[2] / psi[2]
max(TI * eigen(matrix(c(k1, 0, 0, k2), 2, 2) * n)$values)

## [1] 16.25391
```

[Return to task on P136](#)

Solution 92

```

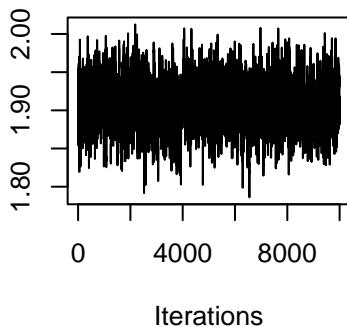
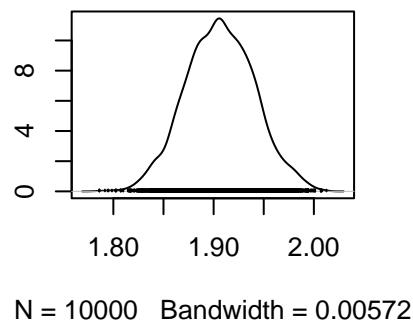
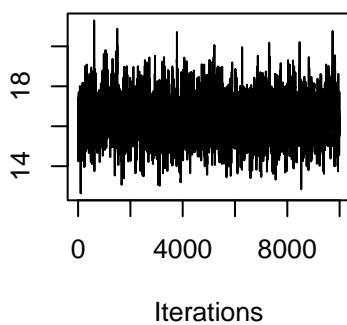
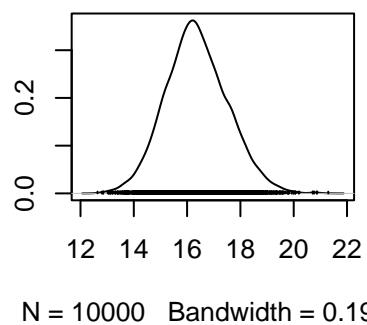
## calculate psi for each posterior sample
psi <- t(apply(as.matrix(twogroup_post), 1, function(x, u, TI) {
    Psi(x, age_widths = c(15, 30), u, TI)
}, u = u, TI = TI))

## infant mixing posterior samples
k1 <- twogroup_post[, 1] / psi[, 1]
k2 <- twogroup_post[, 2] / psi[, 1]
R0_infant <- apply(cbind(k1, k2), 1, function(k, n, TI) {
    max(TI * eigen(matrix(c(k[1], k[2], 0, 0), 2, 2) * n)$values)
}, n = n, TI = TI)

## diagonal mixing MLE
k1 <- twogroup_post[, 1] / psi[, 1]
k2 <- twogroup_post[, 2] / psi[, 2]
R0_diagonal <- apply(cbind(k1, k2), 1, function(k, n, TI) {
    max(TI * eigen(matrix(c(k[1], 0, 0, k[2]), 2, 2) * n)$values)
}, n = n, TI = TI)

## plot and summarise
R0 <- as.mcmc(cbind(R0_infant, R0_diagonal))
plot(R0)

```

Trace of R0_infant**Density of R0_infant****Trace of R0_diagonal****Density of R0_diagonal**

```
summary(R0)
```

```

## 
## Iterations = 1:10000
## Thinning interval = 1

```

```
## Number of chains = 1
## Sample size per chain = 10000
##
## 1. Empirical mean and standard deviation for each variable,
##     plus standard error of the mean:
##
##           Mean      SD  Naive SE Time-series SE
## R0_infant    1.907 0.03405 0.0003405      0.001079
## R0_diagonal 16.360 1.16207 0.0116207      0.035975
##
## 2. Quantiles for each variable:
##
##           2.5%    25%    50%    75%   97.5%
## R0_infant    1.84   1.883   1.906   1.93   1.976
## R0_diagonal 14.18  15.577  16.311  17.11  18.735
```

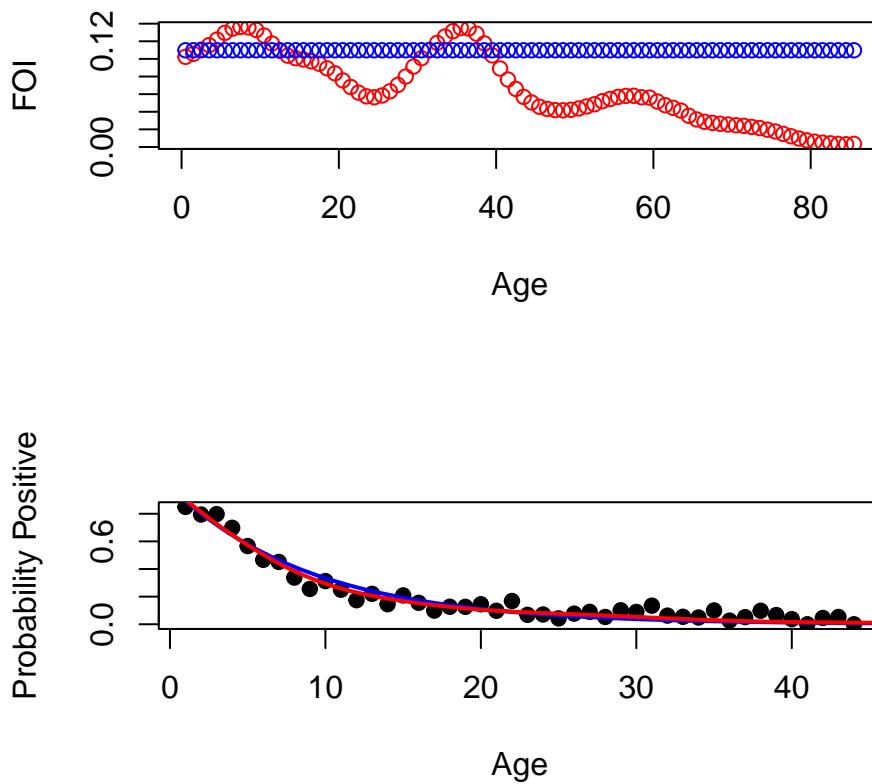
[Return to task on P136](#)

Solution 93

```

POLYMOD_mle <- optimise(
  NextGen_loglik,
  c(1, 15),
  matrixfunction = POLYMODMatrix,
  sero_data = rubella,
  maximum = TRUE
)
POLYMOD_foi = NextGen_foi(POLYMODMatrix(exp(POLYMOD_mle$maximum)))
par(mfrow = c(2, 1))
plot(
  seq(0.5, 86, 1),
  POLYMOD_foi,
  type = 'b',
  xlab = 'Age',
  ylab = 'FOI',
  col = 'red'
)
lines(seq(0.5, 86, 1), homogeneous_foi, type = 'b', col = 'blue')
plot(
  rubella$Age,
  rubella$p_s,
  pch = 19,
  xlab = 'Age',
  ylab = 'Probability Positive'
)
lines(
  seq(0.5, 86, 1),
  piecewise_s(seq(0.5, 86, 1), homogeneous_foi),
  lwd = 2,
  col = 'blue'
)
lines(
  seq(0.5, 86, 1),
  piecewise_s(seq(0.5, 86, 1), POLYMOD_foi),
  lwd = 2,
  col = 'red'
)
par(mfrow = c(1, 1))

```



[Return to task on P141](#)

Answer 94

The homogeneous mixing model has a maximum (log) likelihood value of -136, compared to -117 for the POLYMOD model. On this basis, and from visual inspection of the fit, the POLYMOD mixing model is better supported by the data, although we might want to produce measures such as AIC that penalise for model complexity before drawing firm conclusions.

[Return to task on P142](#)

Answer 95

The POLYMOD mixing assumption estimates a lower basic reproductive ratio than homogeneous mixing, implying that a lower level of vaccination coverage would be necessary to eliminate disease than might be expected under homogeneous mixing.

However, more importantly the POLYMOD model predicts a very different pattern of transmission within the population. The POLYMOD mixing matrix generates two equal peaks in the force of infection that are both higher than the homogeneous mixing estimate. The second peak in young adults is of particular concern for rubella where the disease itself is of less concern than so-called congenital rubella syndrome (CRS)—where infection of pregnant women leads to severe problems for unborn children. Pre-vaccination, with a mean age of infection of ~10 years this second group was relatively unimportant as few susceptible individuals ‘survived’ to that age. But it may become more important after the introduction of vaccination which is likely to increase the mean age of infection into this second core-group. This is the motivation for supplemental vaccination campaigns in teenage girls—i.e. not to reduce the rate of transmission in the community but to directly protect against CRS. The magnitude of this, so-called perverse effect of vaccination, would be greatly underestimated by a (incorrect) assumption of homogeneous mixing.

[Return to task on P142](#)

Solution 96

```

## function to calculate R0 and length of
## infectious periods
R0fn <- function(beta, gamma) {
  data.frame(
    R0 = beta / gamma,
    infperiod = 1 / gamma
  )
}

## summarise approximate posterior
summary(post, transfunc = R0fn)

##
## Iterations = 1:8001
## Thinning interval = 1
## Number of chains = 1
## Sample size per chain = 8001
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##           Mean      SD  Naive SE Time-series SE
## beta     2.9938 0.28385 0.0031733      0.015745
## gamma    1.0457 0.20216 0.0022601      0.010738
## gamma1   0.4608 0.02824 0.0003157      0.001570
## logPost  -67.0245 1.68719 0.0188622      0.140805
## R0       2.9411 0.48789 0.0054544      0.027875
## infperiod 0.9909 0.18449 0.0020626      0.009989
##
## 2. Quantiles for each variable:
##
##           2.5%     25%     50%     75%   97.5%
## beta     2.5032  2.7964  2.9512  3.1595  3.6210
## gamma    0.7229  0.9132  1.0140  1.1655  1.5381
## gamma1   0.4014  0.4410  0.4614  0.4805  0.5113
## logPost  -70.5725 -68.0913 -66.8836 -65.9348 -63.7634
## R0       2.1230  2.5953  2.9134  3.2148  3.9640
## infperiod 0.6502  0.8580  0.9862  1.0950  1.3833

```

[Return to task on P156](#)

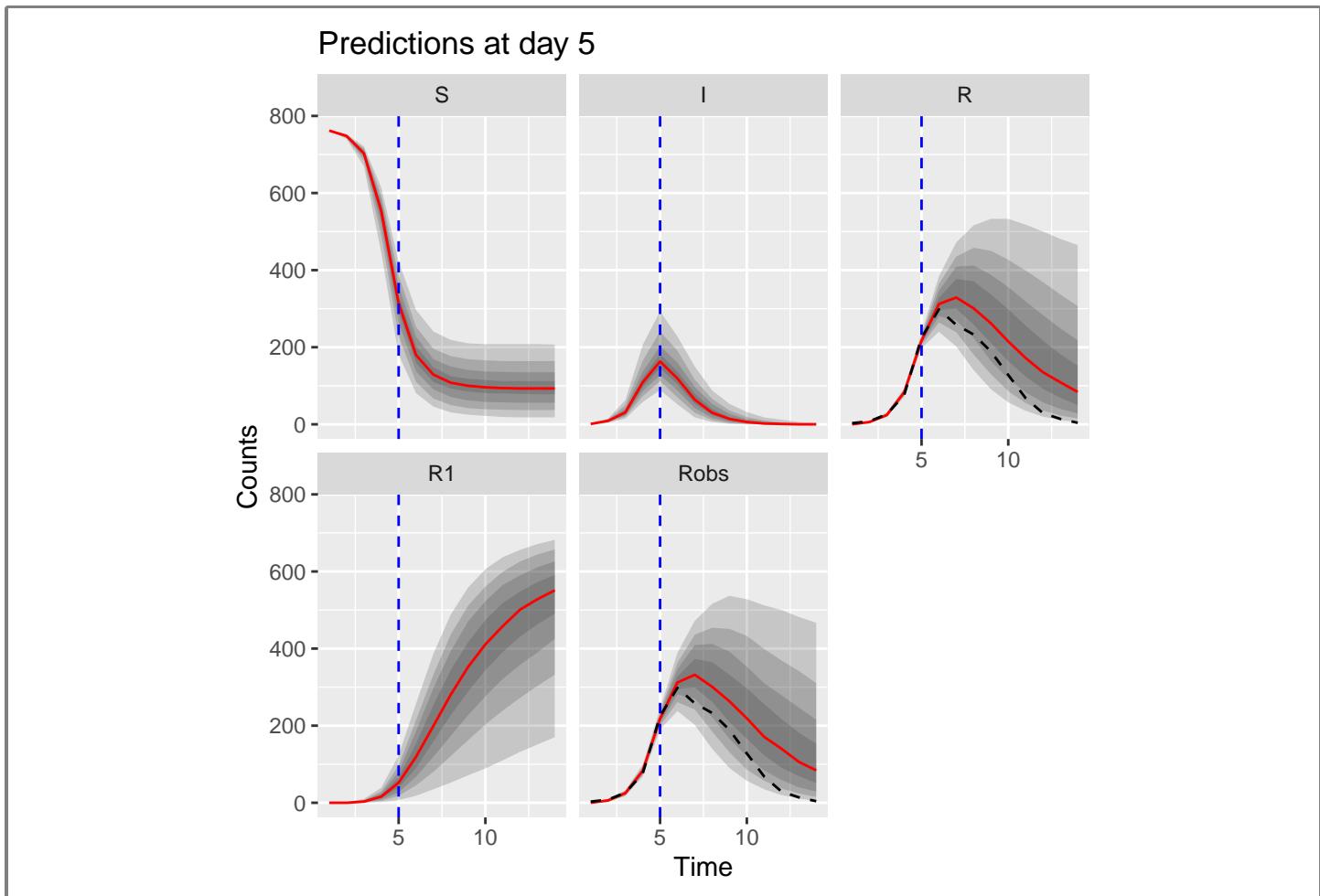
Solution 97

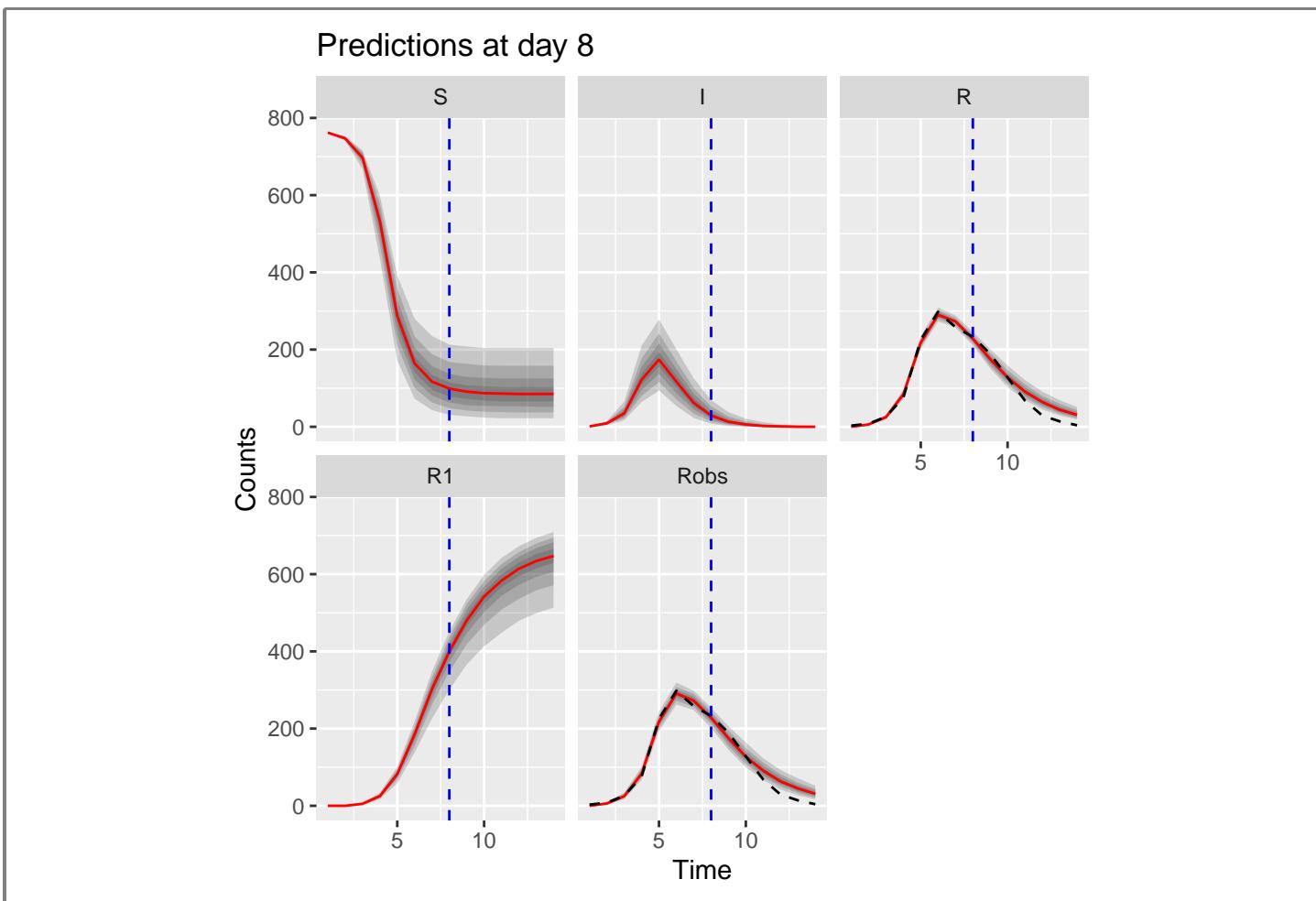
Note that I've suppressed the fitting and plotting code for brevity.

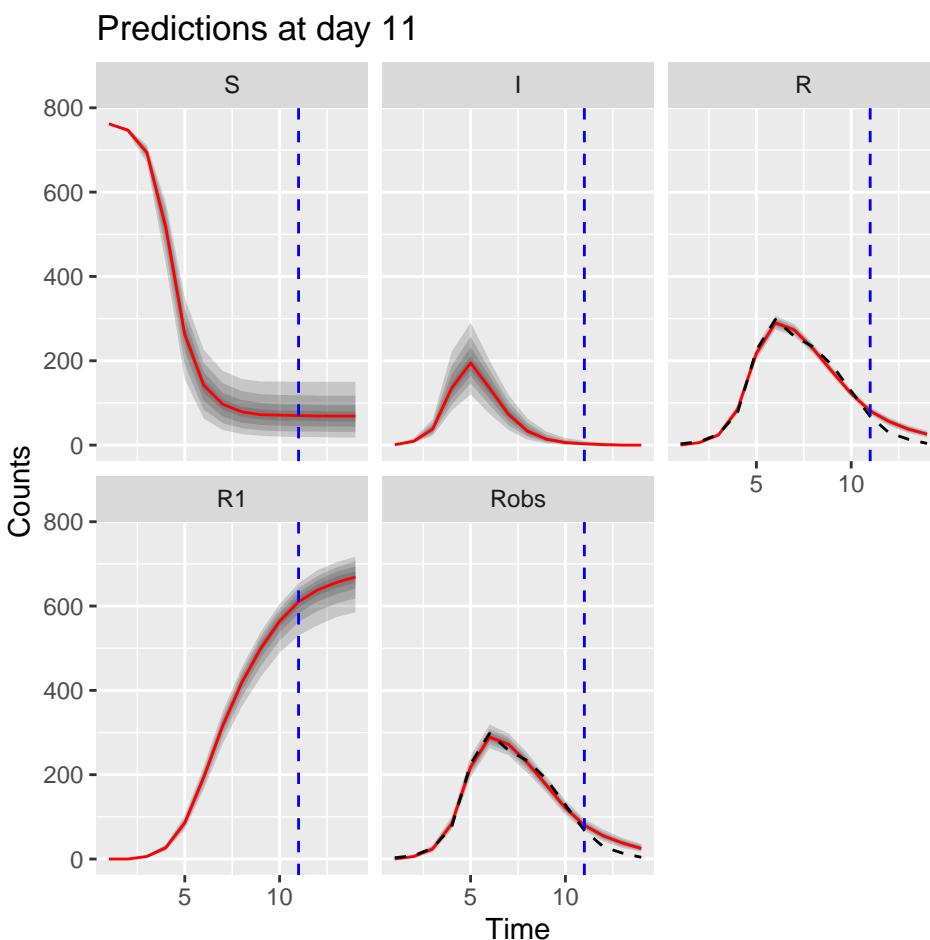
```

## Warning: `cols` is now required when using `unnest()` .
## i Please use `cols = c(value)` .
## `cols` is now required when using `unnest()` .
## i Please use `cols = c(value)` .
## `cols` is now required when using `unnest()` .
## i Please use `cols = c(value)` .

```







We can see that the uncertainties reduce as we fit to more data and as the epidemic begins to die out. We can also see that the actual observed outbreak is within the main uncertainty bounds for each of the model fits, as long as we take account for the observation process. Here the observation process acts mostly as a means of modelling **underreporting**, and hence the **observed** counts tend to be less than the **simulated** counts (shown here by adding the observed data line to both the R and Robs plots). We can also see that the uncertainties in the forecasts get smaller the more data we fit to and also as the epidemic begins to die out.

[Return to task on P159](#)

Solution 98

We have limited the graph to only plot the first 100 cases. We achieved this by using the date of the 100th entry of the line list (`outfluenza1$x`) to set the upper limit.

[Return to task on P169](#)

Solution 99

```
R0final <- function(zf) {
  # zf must be proportion between 0 and 1
  # Trap for division by zero(zf=0.0)
  # Trap for log(1) (-Inf)
  if(zf<=0 || zf>=1.0) {
    return(NA)
  } else {
    return(-log(1-zf)/zf)
  }
}
```

[Return to task on P171](#)

Solution 100

```
R0final(incidence(outfluenza1$x)$n/1300)
```

```
## [1] 2.646094
```

```
R0final(incidence(outfluenza2$x)$n/1300)
```

```
## [1] 2.378981
```

```
R0final(incidence(biggles$x)$n/1300)
```

```
## [1] 2.463143
```

[Return to task on P172](#)

Solution 101

$$R_0 = 1 + 0.335 \times 5 = 2.67$$

[Return to task on P173](#)

Solution 102

$$R_0 = 2.67 \text{ (95\% CI, 2.1–3.1)}$$

[Return to task on P173](#)

Answer 103

We are fitting a log-linear relationship and $\log(0)$ is undefined.

[Return to task on P174](#)

Answer 104

`outflu1.fit2` is clearly a poor fit to the exponential growth rate as we have not subsetted the data to the exponential phase. The point estimates from the incidence package and our manual estimate are pretty comparable, however the confidence intervals from our manual estimate are smaller (0.2–0.4 compared to 0.2–0.5) reflecting the effective **smoothing** we applied to the data by using the cumulative case counts instead of raw incidence. We could similarly smooth the time series by increasing the time interval to sum up incidence - this runs the risk of introducing **artifacts** to the data or giving us false confidence in our estimates.

[Return to task on P176](#)

Solution 106

The confidence intervals from our regression model capture the variability (scatter) in the model output around the best-fit line for this particular realisation of the epidemic model. Stochastic effects will generate far greater variability between realisations, which our linear model contains no information about. The uncertainty in the estimate of the slope will therefore typically underestimate the true variability in the epidemic model. The Likelihood function for the `earlyR` method includes more information on the variability of the serial interval derived from an epidemic model and better represents the true **uncertainty** in our estimates.

[Return to task on P179](#)

Additional information

Relating sticker collecting to the harmonic numbers

In the lecture I claimed that the example of collecting things is somehow related to the harmonic series. To see this does not require sophisticated mathematics.

If there are a total of N stickers, of which you have already collected M , the probability of a new sticker being one you “need” is $p_M = (N - M)/N$. Assuming that successive purchases are independent, the number of stickers that must be bought to get a sticker you need when you already have M stickers in your album would follow a geometric probability distribution^a. The expected number of stickers needed to get the M^{th} sticker is therefore $\frac{1}{p_M}$.

The number of stickers that you must buy to complete the entire album is then just the expected number of stickers to get the first sticker in your album (i.e. when you have already got 0), plus the expected number required for your second sticker (i.e. when you have already got 1), ..., all the way up to the expected number to get the final sticker (i.e. when you have already got $N^{\sim} 1$).

Therefore

$$\text{Average number of stickers required} = \frac{1}{p_0} + \frac{1}{p_1} + \cdots + \frac{1}{p_{N-1}}.$$

However (by the definition of p_M) this reduces to

$$\begin{aligned}\text{Average number of stickers required} &= \frac{N}{N} + \frac{N}{N-1} + \cdots + \frac{N}{1} \\ &= N \left(\frac{N}{N} + \frac{N}{N-1} + \cdots + \frac{1}{1} \right) \\ &= N \left(1 + \cdots + \frac{1}{N-1} + \frac{1}{N} \right) \\ &= N \times H_N\end{aligned}$$

where we reversed the order of the sum in going from line 2 to line 3, and have used the definition of the N^{th} harmonic number (see lecture slides) on line 4.

[Return to P57](#)

^aYou will be reminded of basic facts about probability distributions in a forthcoming lecture. Note the expected number is quite intuitive: as a simple example, consider rolling a dice until you get a six. The probability of getting a six is $\frac{1}{6}$: hopefully it seems reasonable that you should have to roll the dice an average of 6 times to get one. That’s all that is happening here, just written in symbols.

Hint

Remember your log rules, e.g. $\log(xy) = \log(x) + \log(y)$, $\log(\frac{1}{x}) = -\log(x)$ and $\log(e^x) = x$.

[Return to P129](#)

Hints

- Assume that all individuals are susceptible at birth ($p_0 = 1$ for first age cohort).
- Think carefully about what the estimated force-of-infection for the final age cohort should be.
- You may wish to write a loop, or consider using vectors (e.g. `1:(nrow(rubella) - 1)`) to access the different elements of `rubella$p_s`.

[Return to P129](#)

The optim() function

`optim()` is an in-build optimisation function in R. It is used when you want to optimise a function with respect to *multiple* parameters. If you want to optimise a function with respect to a *single* parameter, then use `optimise()` (which we will use later on).

Arguments for `optim()` can be found from the help file in the usual way e.g. `?optim`. The key ones here are:

- `par`: a vector of initial values (for λ_1 and λ_2 here).
- `fn`: a function to be minimised (or maximised), with first argument the vector of parameters over which optimisation is to take place. It should return a scalar result. Here we pass it the `FOI_loglik()` function we defined above.
- ...: additional arguments required for `FOI_loglik()` can be passed to `optim()`; here we pass `foi_key` and `sero_data` arguments.
- `control`: a list of control parameters. Here we set `fnscale = -1` to ensure **maximisation** (else it defaults to minimisation).

Please see the help file for more details.

[Return to P132](#)

A note on priors

The function `MCMCmetrop1R()` provided in the `MCMCpack` library takes a function that returns the un-normalised log-posterior density, that is:

$$\log [f(\mathbf{y} | \boldsymbol{\theta})] + \log [f(\boldsymbol{\theta})]. \quad (13.1)$$

We implement this using the `FOI_loglik()` function defined earlier. If we only export the log-likelihood from this function, then this is equivalent to putting completely flat (so-called improper) priors on the parameters. A subtlety here is that `FOI_loglik()` actually returns `NA` if any of the parameters are **negative**, meaning that in fact we are putting flat priors on λ_1 and λ_2 that are constrained in the region $(0, \infty)$. This is important since we know that rates can't be negative.

If we want to use more structured prior distributions, then we would have to amend `FOI_loglik()` to return (13.1) with the correct prior densities built-in.

Important: you must know what priors you are using, since they are an integral part of a Bayesian analysis.

[Return to P134](#)

Hint

Plotting the fitted model object will give the trace plots, and summarising the model object gives useful summary measures.

[Return to P134](#)

Hint

You will need to use the `max()`, `eigen()` and `matrix()` functions. Remember that here the NGM is $T_I \beta \mathbf{n}$, where \mathbf{n} is the proportion of the population in each age-class, and T_I is the average length of the infectious period.

[Return to P136](#)

Hint

Convert the `twogroups_post` into a matrix using `as.matrix()`, and then either loop over the samples, or use `apply()` to generate posterior samples for ψ_1 and ψ_2 . These can then be converted into posterior samples for k_1 and k_2 and correspondingly into posterior samples for R_0 . `as.mcmc()` can be used to turn `matrix` objects into `mcmc` objects for ease of plotting and summarisation (or just work with the matrix objects directly).

[Return to P136](#)

The optimise() function

We used `optim()` earlier to optimise a function with respect to *multiple* parameters. If you want to optimise a function with respect to a *single* parameter, then we use `optimise()` instead. This works in a similar way, but has slightly different arguments. Most notably, rather than giving initial values for the parameter, we instead define an upper and lower bound over which to search for a maximum or minimum.

Arguments for `optimise()` can be found from the help file in the usual way e.g. `?optimise`. The key ones here are:

- `f`: a function to be minimised (or maximised), with first argument the vector of parameters over which optimisation is to take place. It should return a scalar result. Here we pass it the `NextGen_loglik()` function we defined above.
- `interval`: a vector containing the end-points of the interval to be searched for the minimum (or maximum).
- `...`: additional arguments required for `NextGen_loglik()` can be passed to `optimise()`; here we pass `matrixfunction` and `sero_data` arguments.
- `maximum`: a logical defining whether we **maximise** (`TRUE`) or **minimise** (`FALSE`).

Please see the help file for more details.

[Return to P140](#)

Note on Poisson assumptions

The idea of putting some form of Poisson noise on the parameters of time-series count models has been used various times in the literature (e.g. [Funk et al. 2016](#)). Often the observation term is placed on the *incidence* curves (i.e. new infections / removals). The variance of the Poisson distribution is equal to the mean (so a mean of R'_t has a variance of R'_t also), and since incidence curves tend to be smaller in magnitude than the counts in each class, then this can result in a tighter match between the simulations and the data. It is often usual that the mean of the Poisson noise is also scaled by some parameter corresponding to an *under-reporting* rate, meaning that the model assumes that the average observed counts are less than the true values.

In this example, we do not have *incidence* curves, so we place a Poisson error around the R'_t counts instead. Note here that there is no strong epidemiological reason for the Poisson term in this case. We can assume that in a closed boarding school population, it would be unlikely that too many ill boys would be missed, and hence the observed counts are likely to be close to the true counts, and thus the amount of variation added by the Poisson noise term

here is likely to be larger than we might ideally think is present in the data. In essence the Poisson term is providing an **approximation**, allowing simulations to have a non-negligible weight even if they do not match the observed curves exactly. As such the introduction of this term in this instance is introducing some approximation into the process, but in doing so is allowing us to fit a model in a computationally feasible way. In practice I would want to use a distribution with lower variances however.

To this end, other options might include putting a Gaussian error around the observed counts, where the variance could be made smaller than the mean (e.g. $R_t \sim N(R'_t, \alpha R'_t + 1)$ for $\alpha \in (0, 1)$). This would penalise simulations that lie further away from the observed data more than the Poisson model we are using, at the cost of requiring a larger number of particles / a longer run time to evaluate. (Bear in mind also that the Gaussian models *continuous* random variables, and is unbounded. Hence if you simulate from this observation process then you could get simulated counts that are both non-integer and negative-valued. This type of approximation might help model fitting, but is a good example of where the biological interpretation of the noise process is not always realistic, but might be “good enough.”)

Hence, other (better) observation processes could also be used, but this choice allows us to produce some reasonable estimates in a reasonable computational load.

[Return to P146](#)

Hint

Pass a `transfunc` argument to the `summary()` function. Take a look at the help file for the `summary()` method for `PMCMC` objects i.e. `?summary.PMCMC`.

[Return to P156](#)

Standard *SEIR* (exponential) with sinusoidal forcing

We first present an implementation of the standard *SEIR* model—with constant rates of progression through the latent and infectious compartments—and sinusoidal forcing term. The basic template here is similar to what you have seen before in earlier practicals except that the transmission rate now depends on the time. Here we wrote a wrapper function which sets some sensible default initial conditions (the fixed points of the unforced *SEIR* model where $\alpha = 0.0$) and then runs the model for three periods. No matter the initial model conditions the seasonally forced *SEIR* model will **eventually** converge towards a stable dynamic behaviour (often described as an “attractor” in analogy to the fixed points of an unforced model). We can run the model for a “burn-in” period (analogous to the same concept for MCMC convergence) to remove this “transient” behaviour. Our function solves the model for three periods. We forward-simulate the initial conditions for `burnin` years discarding the results but using the final state to set the initial conditions for a run of `prevacc` years. We then adjust the birth rate to model vaccination at birth and then run for an additional `postvacc` years, returning a tibble with the sampling times and state variables of the model:

```

SEIR_exp <- function(t,y,theta) {

  beta = theta[1];
  TE = theta[2];
  TI = theta[3];
  alpha = theta[4];
  mu = theta[5]

  I_tot=0;
  N_tot=0;

  dy_dt = numeric(4);

  N_tot = sum(y)

  lambda = (1 + alpha * cos(2*pi*t/364))*beta*y[3]/N_tot

  dy_dt[1] = mu*N_tot - (lambda + mu) * y[1];
  dy_dt[2] = lambda * y[1] - (1.0/TE + mu) * y[2];
  dy_dt[3] = (1.0/TE) * y[2] - (1.0/TI + mu) * y[3];
  dy_dt[4] = (1.0/TI)*y[3] - mu*y[4];

  return(list(dy_dt))
}

det_SEIR_exp <-function(N0,R0,TE,TI,mu,alpha,burnin,prevacc,postvacc,vacc)
{

  theta = numeric(7)

  # Set initial conditions to endemic fixed points

  beta = R0/TI
  s0 = 1/R0
  e0 = (TI*mu*(mu+1/TI)/(beta/TE))*(R0-1)
  i0 = (mu/beta)*(R0-1)

  y0 = numeric(4);
  y0[1] = s0*N0;
  y0[2] = e0*N0;
  y0[3] = i0*N0;
  y0[4] = (1-s0-e0-i0)*N0;

  theta[1] = beta;
  theta[2] = TE;
  theta[3] = TI;
  theta[4] = alpha;
  theta[5] = mu

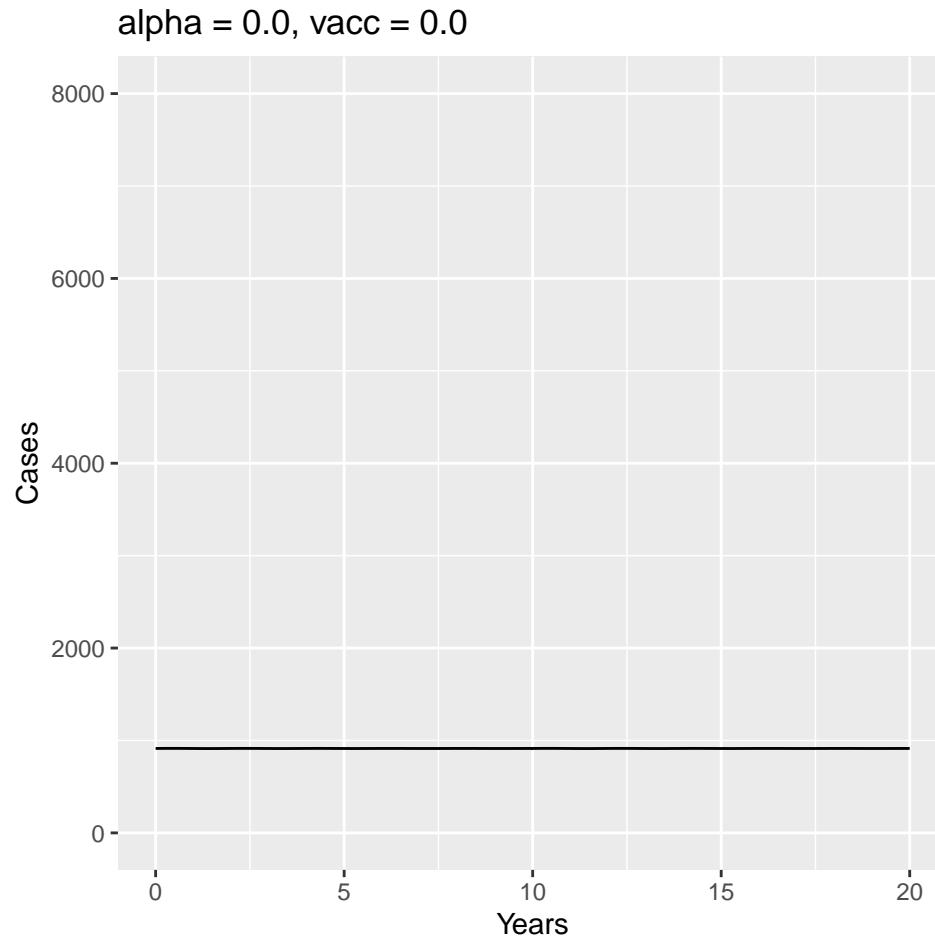
  # Run for burnin years and use final state
  # as initial condition
  y<-lsoda(y0,seq(0,burnin*364,1),SEIR_exp,theta)
  y0 = y[dim(y)[1],2:5]
  # Run for prevacc years
  y<-lsoda(y0,seq(0,prevacc*364,1),SEIR_exp,theta)
  # store time (in years) and numbers in each compartment
  output <- tibble(time=y[,1]/364,S=y[,2],E=y[,3],I=y[,4],R=y[,5])
  # Reduce birth rate by vaccination proportion
  theta[5] = (1-vacc)*mu
  # Run for postvacc years
  y<-lsoda(y0,seq(0,postvacc*364,1),SEIR_exp,theta)
}

```

Using the suggested parameter values you should find that a seasonal forcing of $\alpha = 0.19$ gives a reasonable qualitative fit to the two year cycle of measles outbreaks seen in London after 1950:

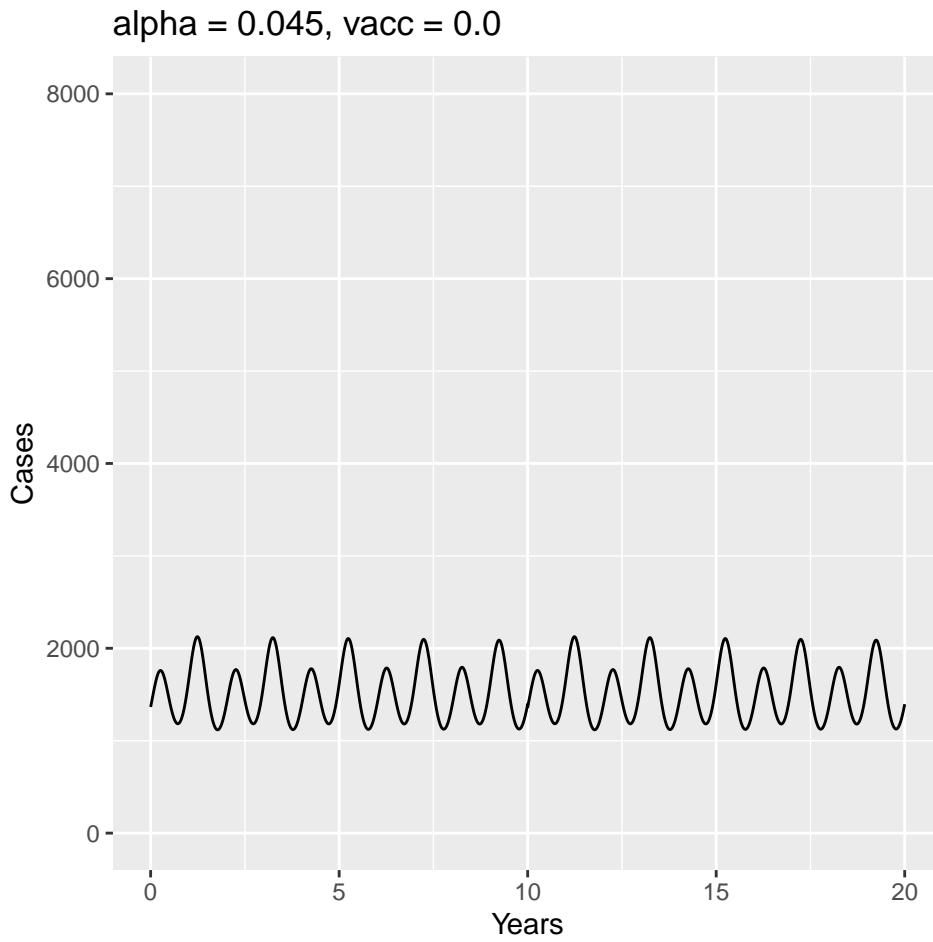
```
x<-det_SEIR_exp(3.3e6,17,8,3,20/(1000*364),0.0,50,10,10,0.0)

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.0, vacc = 0.0") + ylim(0,8000)
```



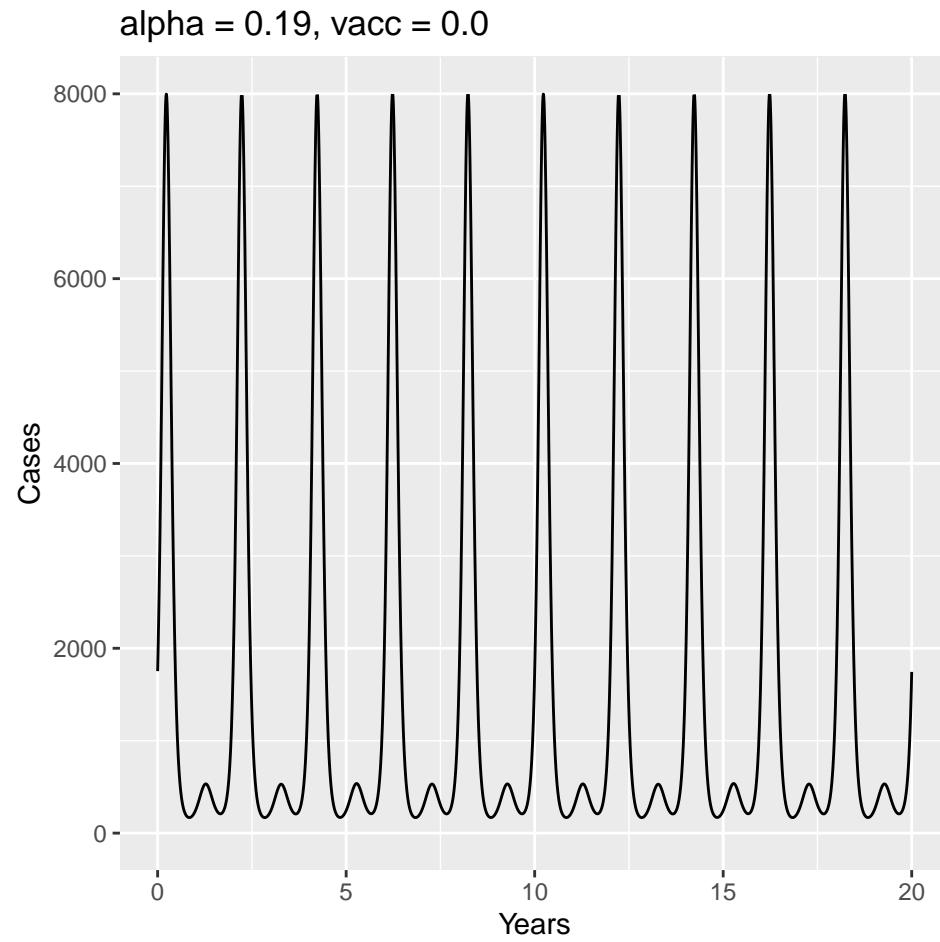
```
x<-det_SEIR_exp(3.3e6,17,8,5,20/(1000*364),0.045,50,10,10,0.0)

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.045, vacc = 0.0") + ylim(0,8000)
```



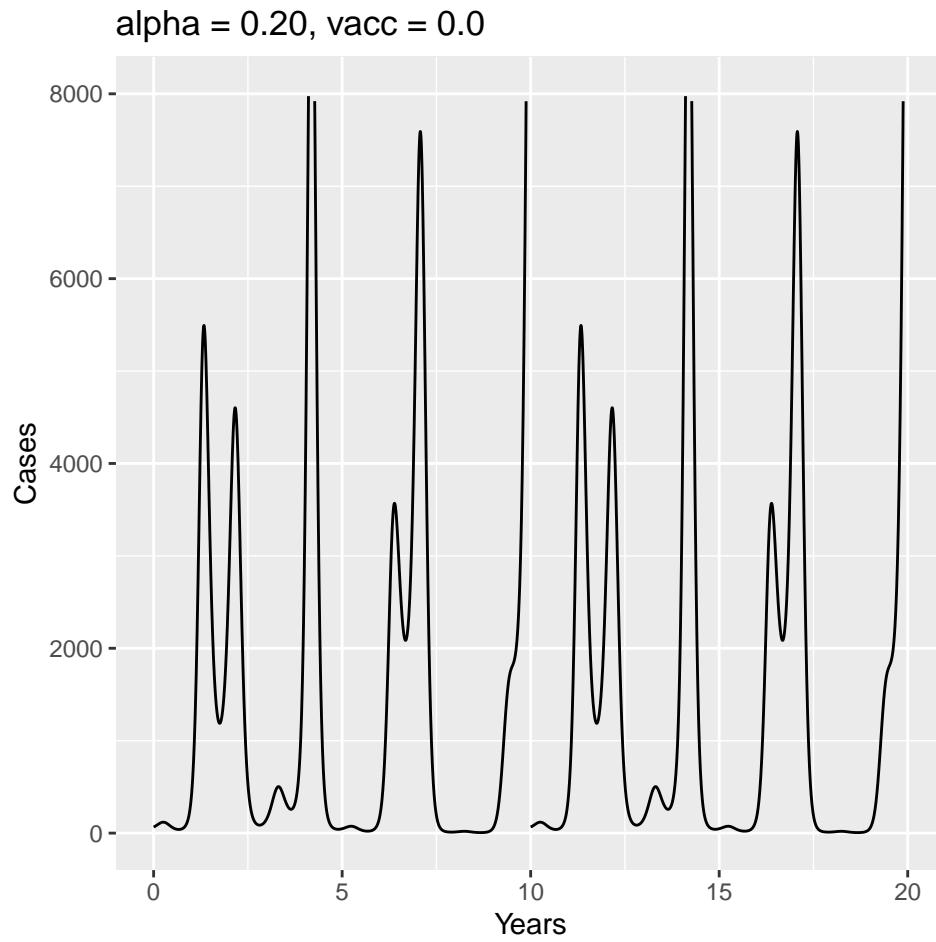
```
x<-det_SEIR_exp(3.3e6,17,8,5,20/(1000*364),0.19,50,10,10,0.0)

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.19, vacc = 0.0") + ylim(0,8000)
```



```
x<-det_SEIR_exp(3.3e6,17,8,5,20/(1000*364),0.20,50,10,10,0.0)

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.20, vacc = 0.0") + ylim(0,8000)
```

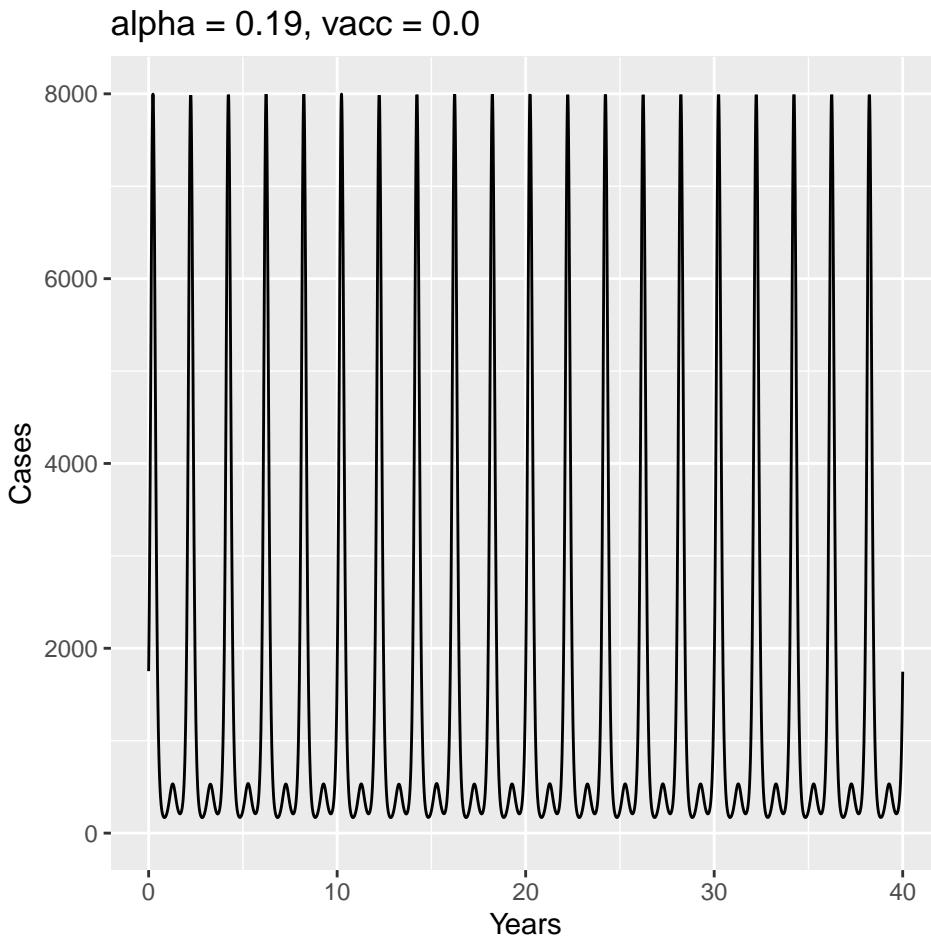


For sinusoidal forcing the bifurcation to two-yearly cycles occurs for a very small amount of seasonal forcing ($\alpha \sim 0.02$) with the amplitude of the two-year cycle increasing with increasing α before further bifurcations to three, four and irregular multiannual cycles.

If we now examine the effect of increasing vaccination coverage we see that at low coverage vaccination progressively increases the time between outbreaks—shifting cycles to 3, 4 and higher (irregular cycles). However, the incidence between outbreaks now falls to unrealistically low levels (nano-scale number of infectives!)—highlighting the likely increased importance of stochastic effects after the introduction of vaccination.

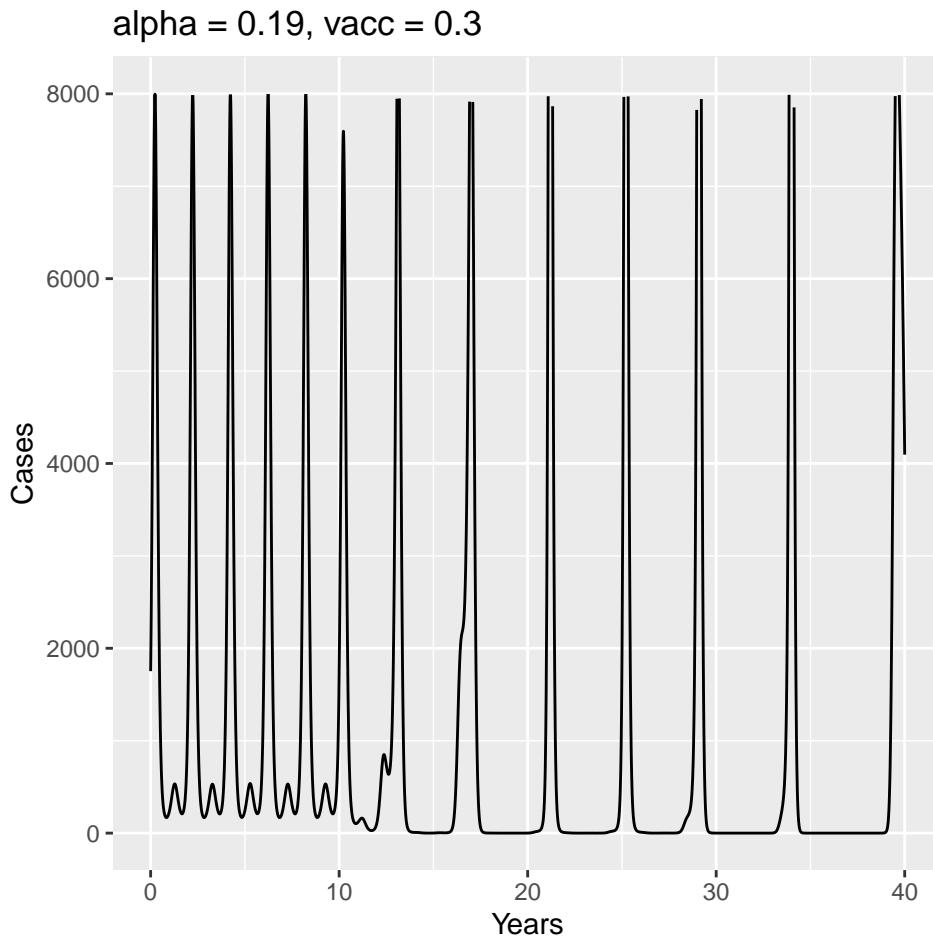
```
x<-det_SEIR_exp(3.3e6,17,8,5,20/(1000*364),0.19,50,10,30,0.0)

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.19, vacc = 0.0") + ylim(0,8000)
```



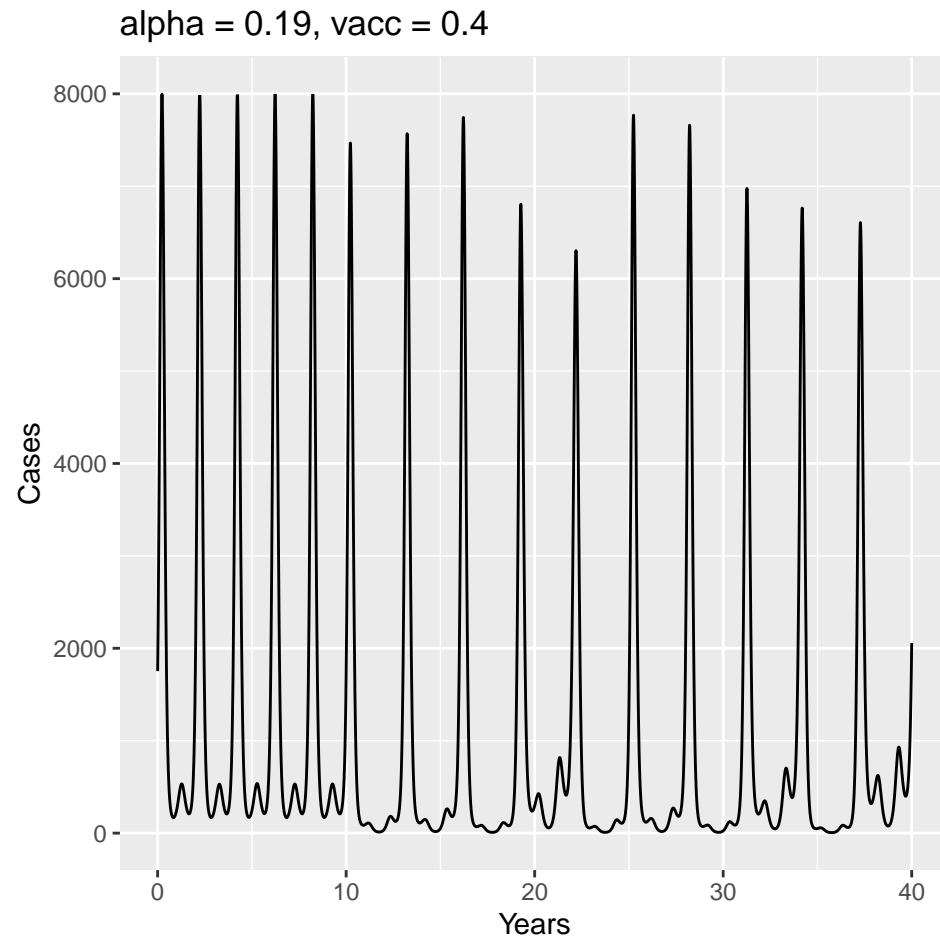
```
x<-det_SEIR_exp(3.3e6,17,8,5,20/(1000*364),0.19,50,10,30,0.3)

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.19, vacc = 0.3") + ylim(0,8000)
```



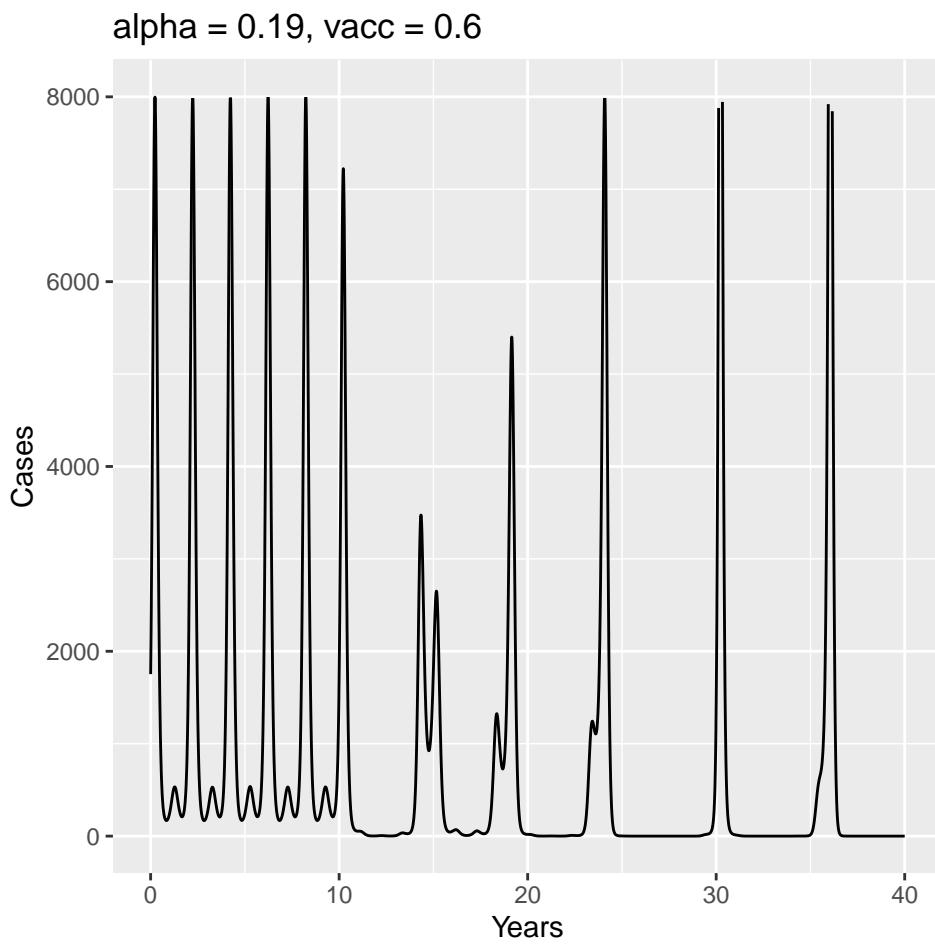
```
x<-det_SEIR_exp(3.3e6,17,8,5,20/(1000*364),0.19,50,10,30,0.4)

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.19, vacc = 0.4") + ylim(0,8000)
```



```
x<-det_SEIR_exp(3.3e6,17,8,5,20/(1000*364),0.19,50,10,30,0.6)

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.19, vacc = 0.6") + ylim(0,8000)
```



These types of models can be very sensitive to both initial conditions and numerical errors meaning that more sophisticated numerical methods (such as AUTO) need to be used to map out the full range of dynamical behaviours accurately.

[Return to P187](#)

Standard *SEIR* (exponential) with term-time forcing

We now adapt the previous code to use a term-time forcing function and run the same model scenarios:

```

SEIR_exp <- function(t,y,theta) {

  beta = theta[1];
  TE = theta[2];
  TI = theta[3];
  alpha = theta[4];
  mu = theta[5]

  I_tot=y[3];
  N_tot=0;

  dy_dt = numeric(4);

  N_tot = sum(y)

  terms = mk_terms(beta,alpha)
  lambda = terms[1 + t %% 364]*beta*I_tot/N_tot

  dy_dt[1] = mu*N_tot - (lambda + mu) * y[1];
  dy_dt[2] = lambda * y[1] - (1.0/TE + mu) * y[2];
  dy_dt[3] = (1.0/TE) * y[2] - (1.0/TI + mu) * y[3];
  dy_dt[4] = (1.0/TI)*y[3] - mu*y[4];

  return(list(dy_dt))
}

```

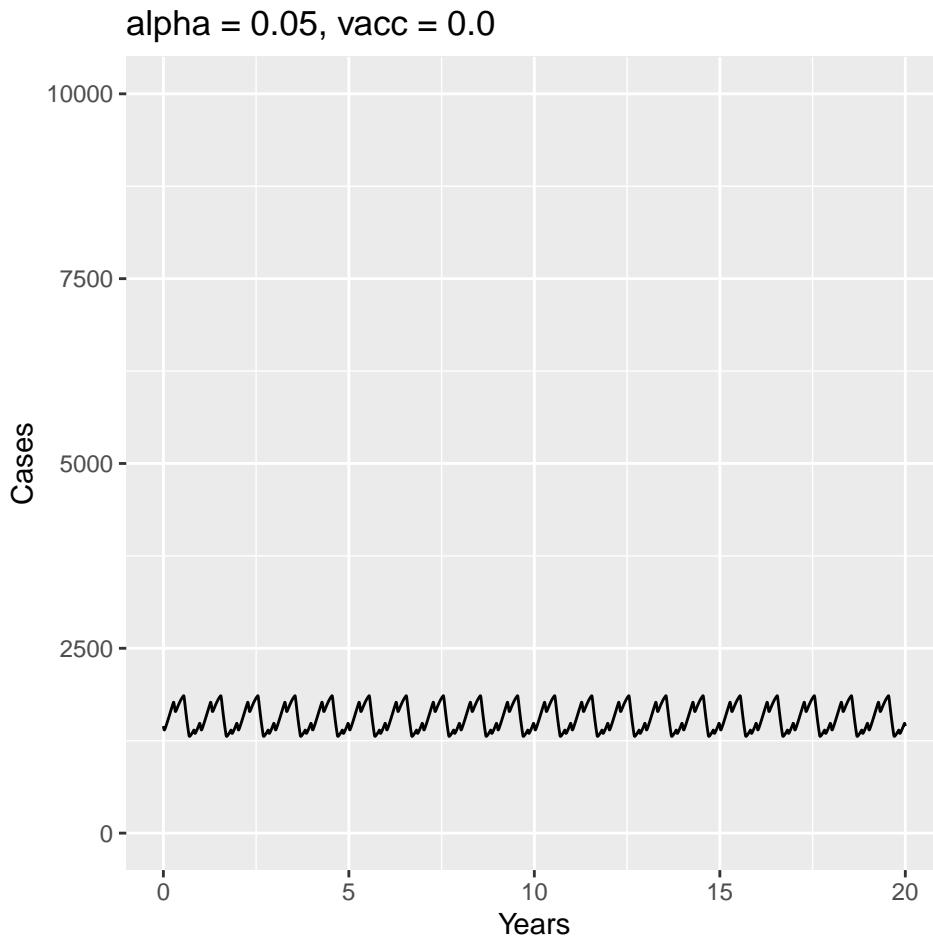
The term-time forcing function leads to a very different (and richer) bifurcation structure. Indeed, if you tried this form of the model you likely could not find values of α that gave a reasonable match to the London time-series for the assumed parameter values. Reducing the assumed value of R_0 to 12 you can achieve a comparable fit (arguably better with respect to the qualitative shape of the attractor).

```

x<-det_SEIR_exp(3.3e6,12,8,5,20/(1000*364),0.05,50,10,10,0.0)

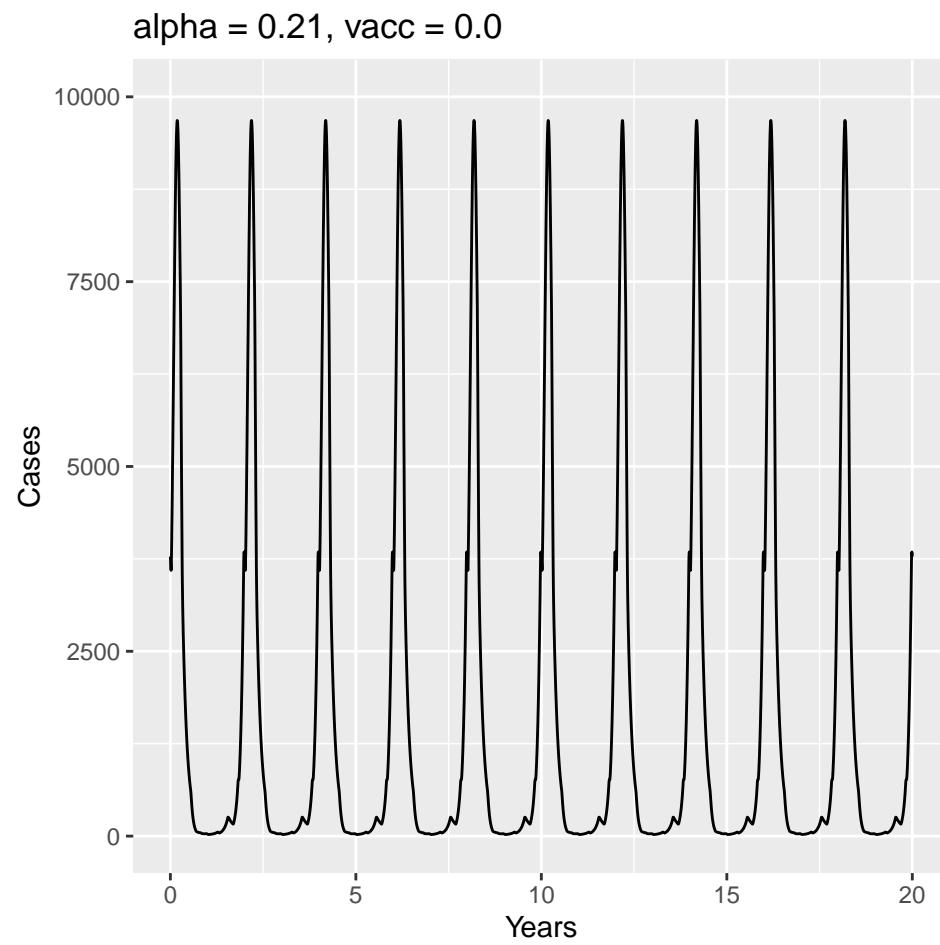
ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.05, vacc = 0.0") + ylim(0,10000)

```



```
x<-det_SEIR_exp(3.3e6,12,8,5,20/(1000*364),0.21,50,10,10,0.0)

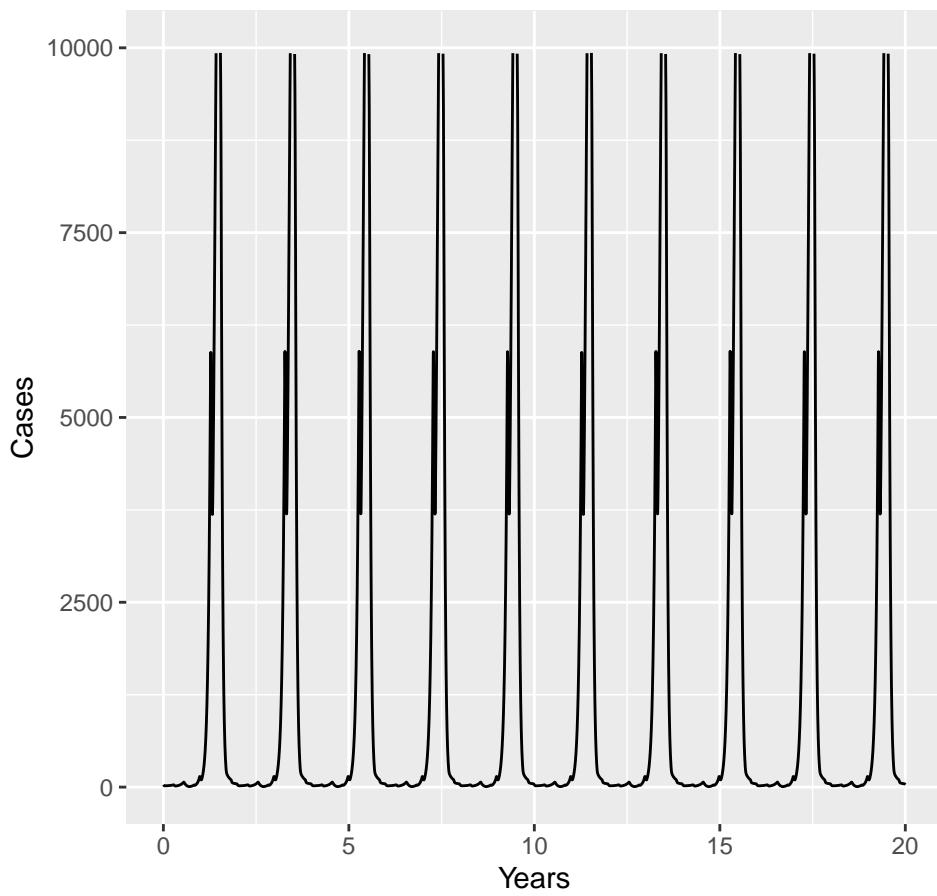
ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.21, vacc = 0.0") + ylim(0,10000)
```



```
x<-det_SEIR_exp(3.3e6,12,8,5,20/(1000*364),0.40,50,10,10,0.0)

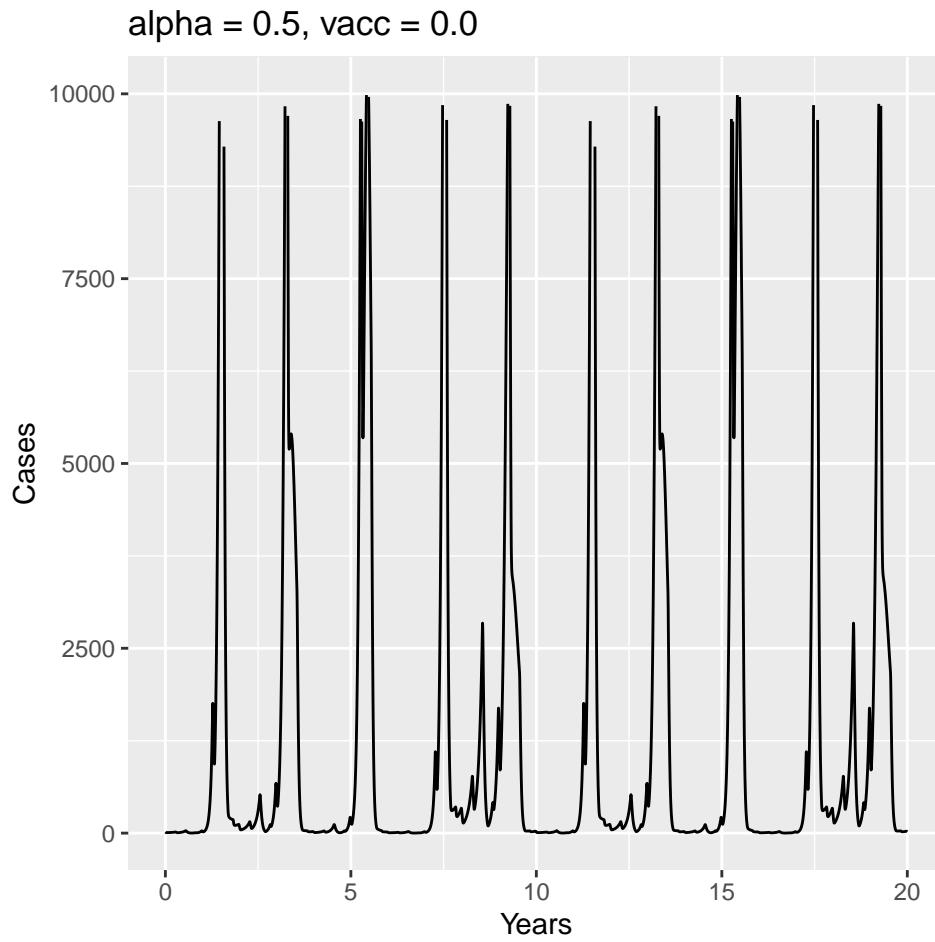
ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.4, vacc = 0.0") + ylim(0,10000)
```

alpha = 0.4, vacc = 0.0



```
x<-det_SEIR_exp(3.3e6,12,8,5,20/(1000*364),0.5,50,10,10,0.0)

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.5, vacc = 0.0") + ylim(0,10000)
```

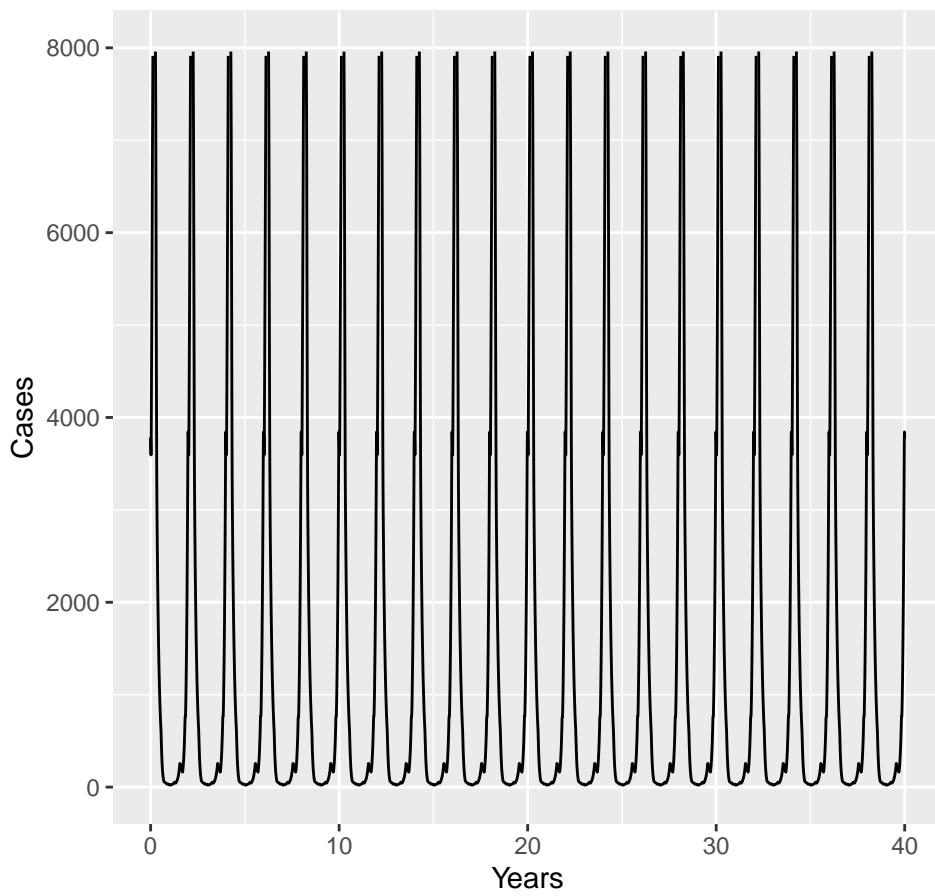


We see the same qualitative impact of vaccination as before, but note that the higher troughs in prevalence between major outbreaks predicted by the term-time forcing model mean that stable (and plausible in terms of depth of trough between outbreaks) dynamics are seen for a wider range of birth/vaccination rates:

```
x<-det_SEIR_exp(3.3e6,12,8,5,20/(1000*364),0.21,50,10,30,0.0)

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.21, vacc = 0.0") + ylim(0,8000)
```

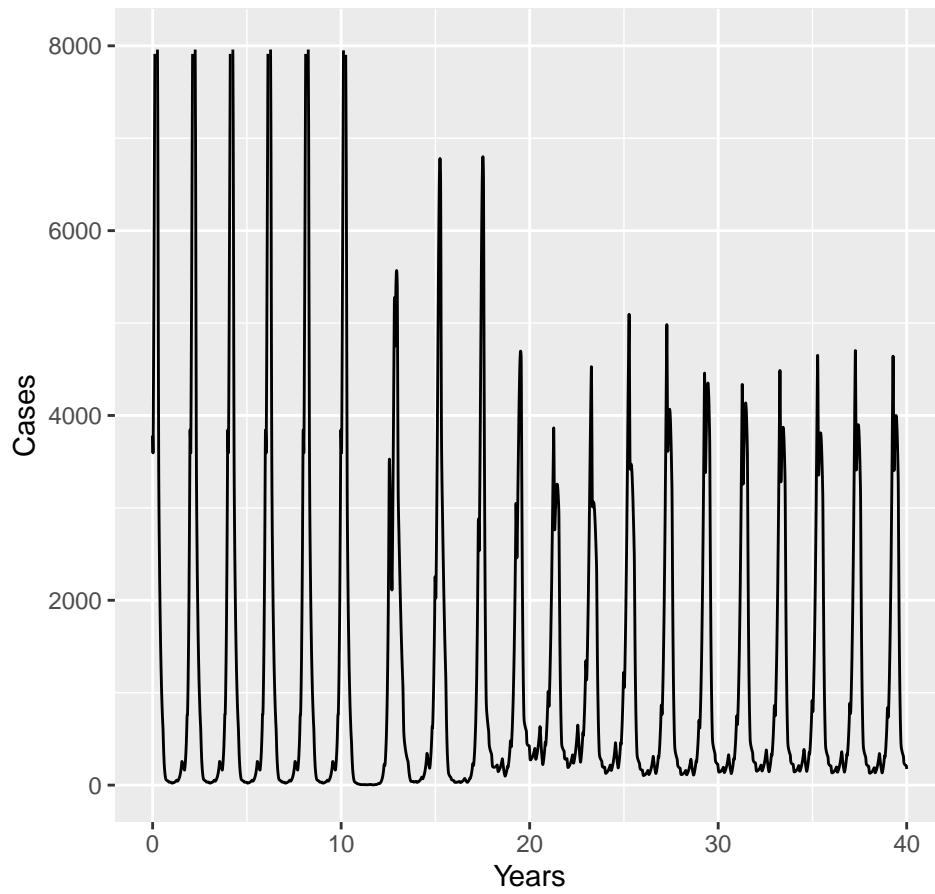
alpha = 0.21, vacc = 0.0



```
x<-det_SEIR_exp(3.3e6,12,8,5,20/(1000*364),0.21,50,10,30,0.3)

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.21, vacc = 0.3") + ylim(0,8000)
```

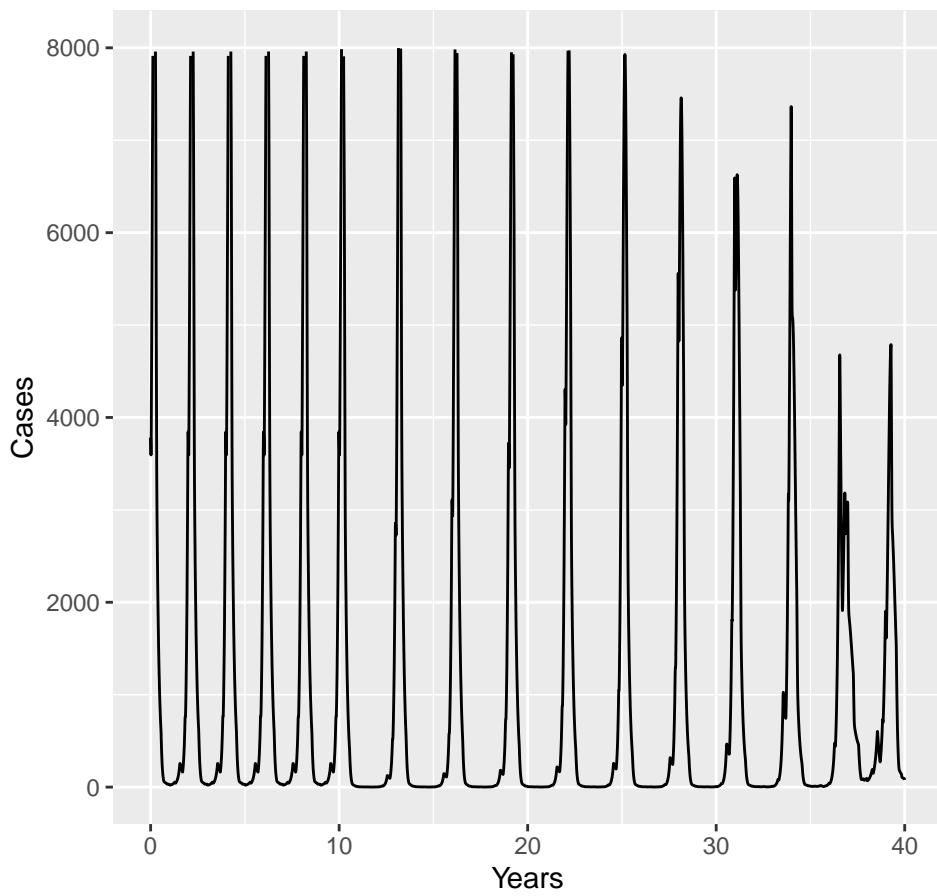
alpha = 0.21, vacc = 0.3



```
x<-det_SEIR_exp(3.3e6,12,8,5,20/(1000*364),0.21,50,10,30,0.4)

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.21, vacc = 0.4") + ylim(0,8000)
```

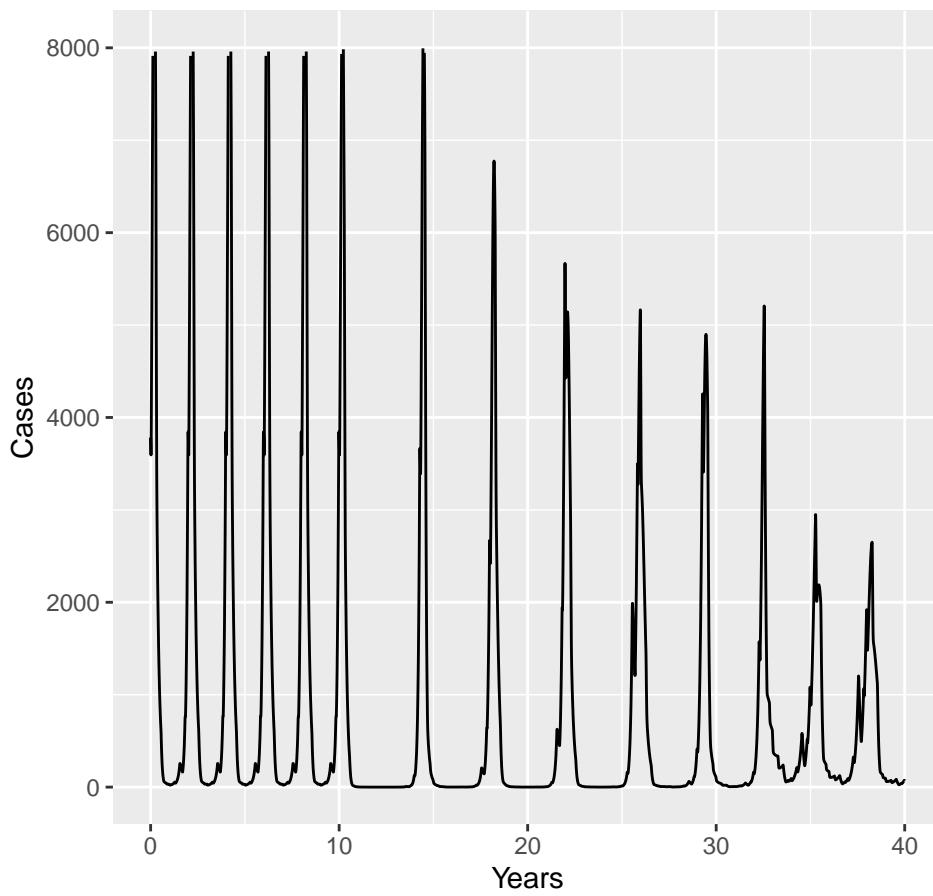
alpha = 0.21, vacc = 0.4



```
x<-det_SEIR_exp(3.3e6,12,8,5,20/(1000*364),0.21,50,10,30,0.6)

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.21, vacc = 0.6") + ylim(0,8000)
```

alpha = 0.21, vacc = 0.6



[Return to P187](#)

Gamma *SEIR* with term-time forcing

Finally, we present an implementation of the term-time forcing model with realistic (gamma) distributed latent and infectious periods. The Gamma-*SEIR* model is much more sensitive to changes in the forcing amplitude α requiring lower amplitudes to generate the same qualitative dynamics as the (less biologically accurate) standard model with constant rates. These models are even more difficult to work with numerically and presented here purely for completeness and a cautionary note on the extent to which these model assumptions impact on the range of model parameters that will be consistent with real data. This sensitivity, and strong trade-offs between transmission rates, birth rates and distributional assumptions makes this type of model particular problematic (and therefore particularly interesting) to perform inference with.

```

SEIR_gamma <- function(t,y,theta) {

  x_i = integer(2);
  beta = theta[1];
  TE = theta[2];
  TI = theta[3];
  alpha = theta[4];
  mu = theta[5]
  x_i[1] = theta[6];
  x_i[2] = theta[7];

  I_tot=0;
  N_tot=0;

  dy_dt = numeric(2+x_i[1]+x_i[2]);

  for(k in (1+x_i[1]+1):(1+x_i[1]+x_i[2])){I_tot = I_tot + y[k];}
  for(k in 1:(2+x_i[1]+x_i[2])){N_tot = N_tot + y[k];}

  terms = mk_terms(beta,alpha)
  lambda = terms[1 + t %% 364]*beta*I_tot/N_tot

  #Susceptibles
  dy_dt[1] = mu*N_tot - (lambda + mu) * y[1];
  #First stage of Exposed (but not yet infectious)
  dy_dt[2] = lambda * y[1] - (x_i[1]/TE + mu) * y[2];

  #If shape parameter > 1 update internal exposed stages
  if(x_i[1]>1){
    for(k in 3:(1+x_i[1]))
      {dy_dt[k] = (x_i[1]/TE) * y[k-1] - (x_i[1]/TE + mu) * y[k];}
  }
  #First infectious stage
  dy_dt[(1+x_i[1]+1)] = (x_i[1]/TE)*y[(1+x_i[1])] - (x_i[2]/TI + mu)*y[(1+x_i[1]+1)];
  #If more than one infectious stage, run through internal stages
  if(x_i[2]>1)
  {
    for(k in (3+x_i[1]):(1+x_i[1]+x_i[2]))
      {dy_dt[k] = (x_i[2]/TI) * y[k-1] - (x_i[2]/TI + mu) * y[k];}
  }

  # Final absorbing recovered stage
  dy_dt[(1+x_i[1]+x_i[2]+1)] = (x_i[2]/TI)*y[(1+x_i[1]+x_i[2])] - mu*y[(1+x_i[1]+x_i[2]+1)];

  return(list(dy_dt))
}

det_SEIR_gamma <-function(N0,R0,TE,TI,shape_E,shape_I,mu,alpha,burnin,prevacc,postvacc,vacc)

{
  theta = numeric(7)

  # Use fixed points of exponential model to approximate initial conditions
  # (Need to run burn in period anyway)

  beta = R0/TI
  s0 = 1/R0
  e0 = (TI*mu*(mu+1/TI)/(beta/TE))*(R0-1)
  i0 = (mu/beta)*(R0-1)

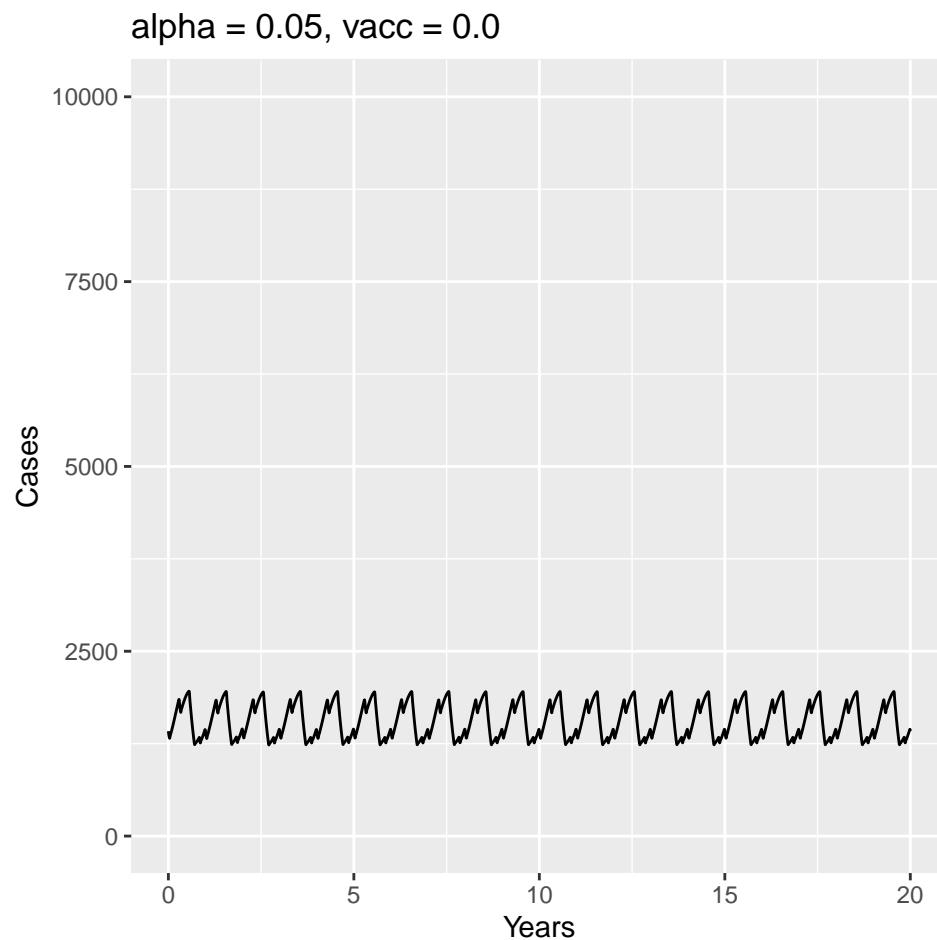
  y0 = numeric(2+shape_E+shape_I);
  y0[1] = s0;
  y0[2] = e0;
  y0[3] = i0;
  y0[4] = 0;
  y0[5] = 0;
  y0[6] = 0;
  y0[7] = 0;
}
```

```
# The extra compartments in gamma-distributed models can cause memory problems
# Here we run the garbage collection function manually to try and head off any crashes

# gc()

x<-det_SEIR_gamma(3.3e6,12,8,5,8,5,20/(1000*364),0.05,50,10,10,0.0)

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.05, vacc = 0.0") + ylim(0,10000)
```

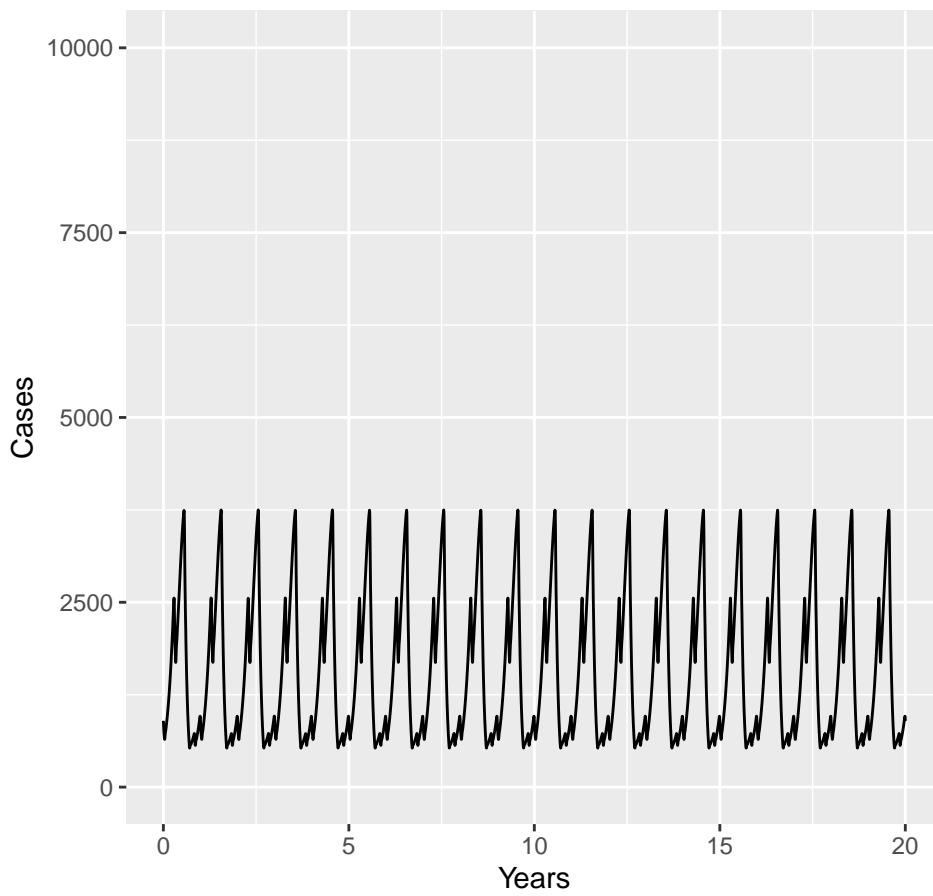


```
x<-det_SEIR_gamma(3.3e6,12,8,5,8,5,20/(1000*364),0.2,50,10,10,0.0)

# gc()

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.2, vacc = 0.0") + ylim(0,10000)
```

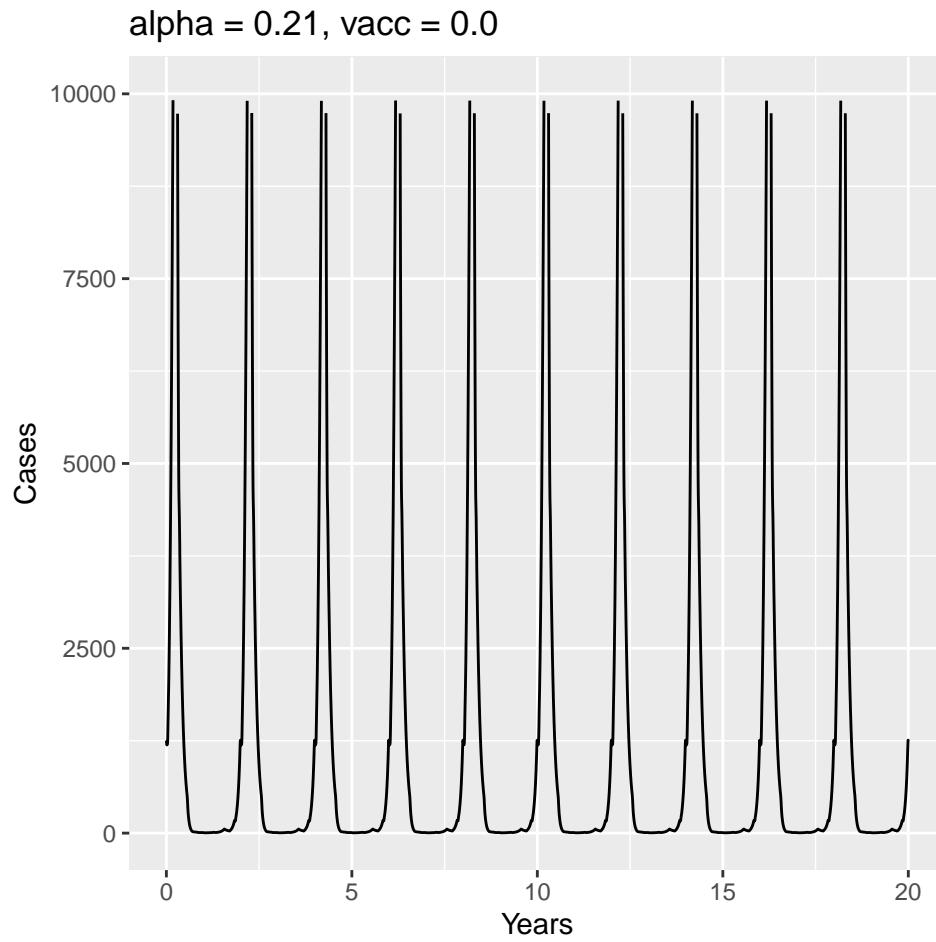
alpha = 0.2, vacc = 0.0



```
x<-det_SEIR_gamma(3.3e6,12,8,5,8,5,20/(1000*364),0.21,50,10,10,0.0)

# gc()

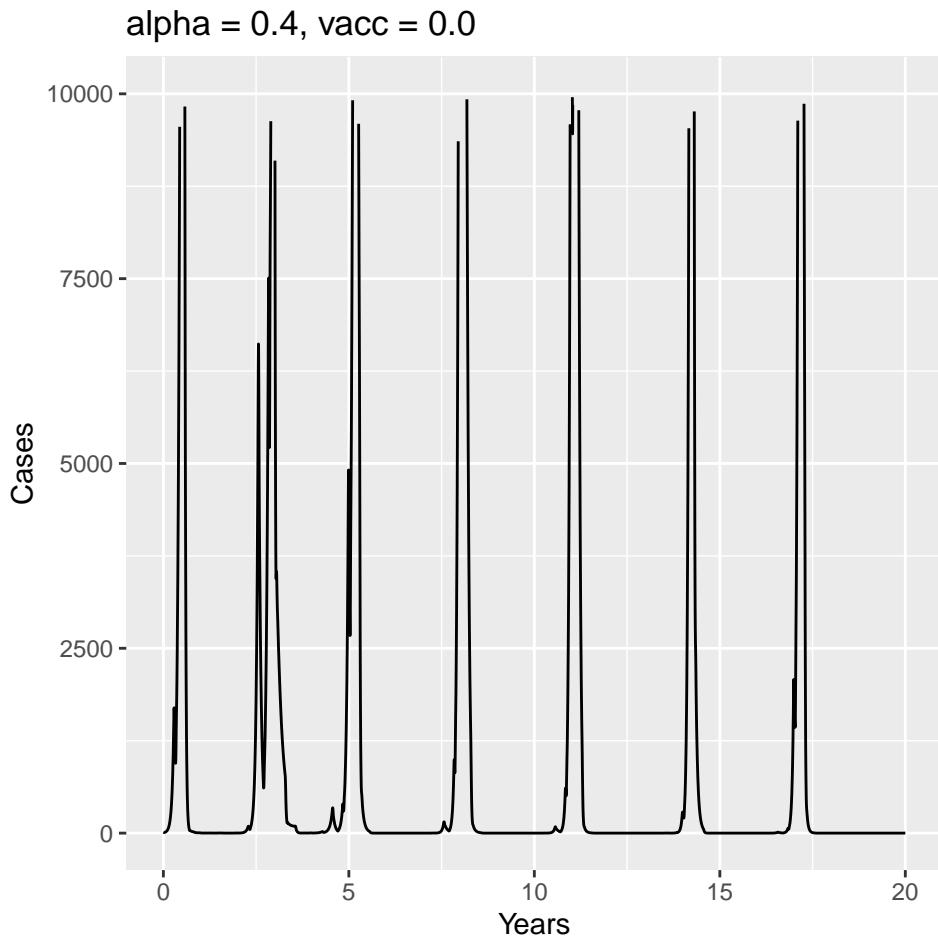
ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.21, vacc = 0.0") + ylim(0,10000)
```



```
x<-det_SEIR_gamma(3.3e6,12,8,5,8,5,20/(1000*364),0.4,50,10,10,0.0)

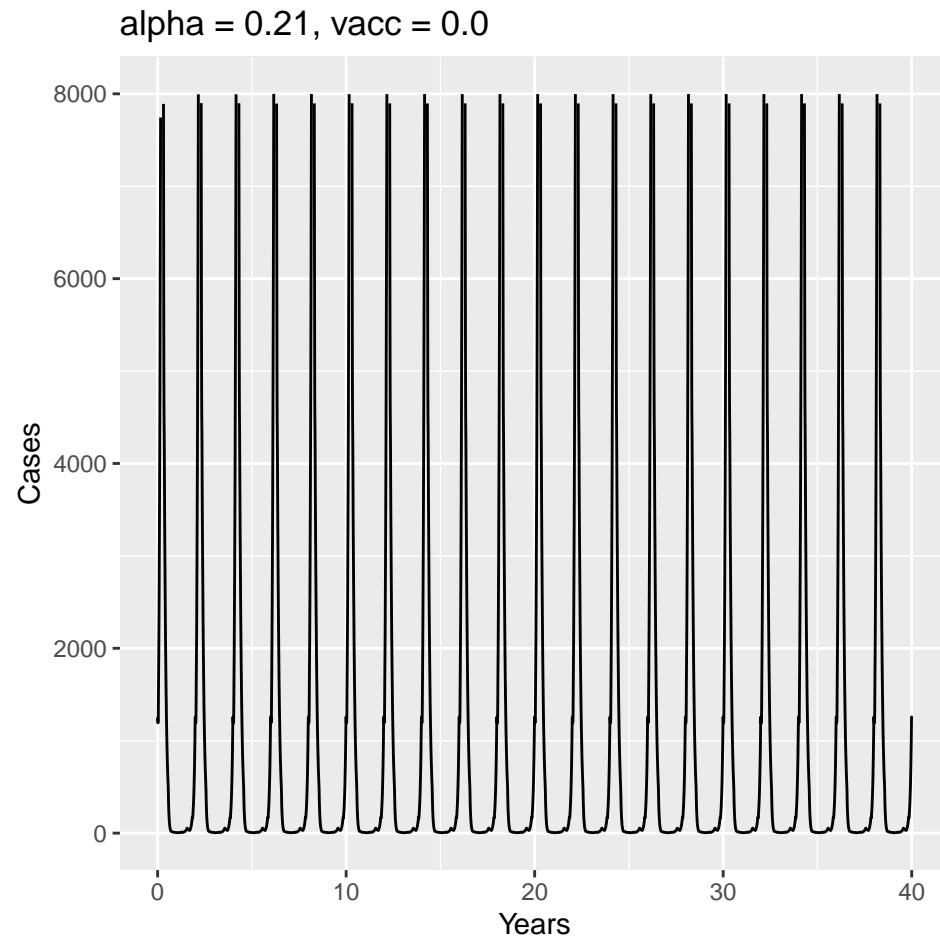
# gc()

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.4, vacc = 0.0") + ylim(0,10000)
```



```
# gc()
x<-det_SEIR_gamma(3.3e6,12,8,5,8,5,20/(1000*364),0.21,50,10,30,0.0)
# gc()

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.21, vacc = 0.0") + ylim(0,8000)
```

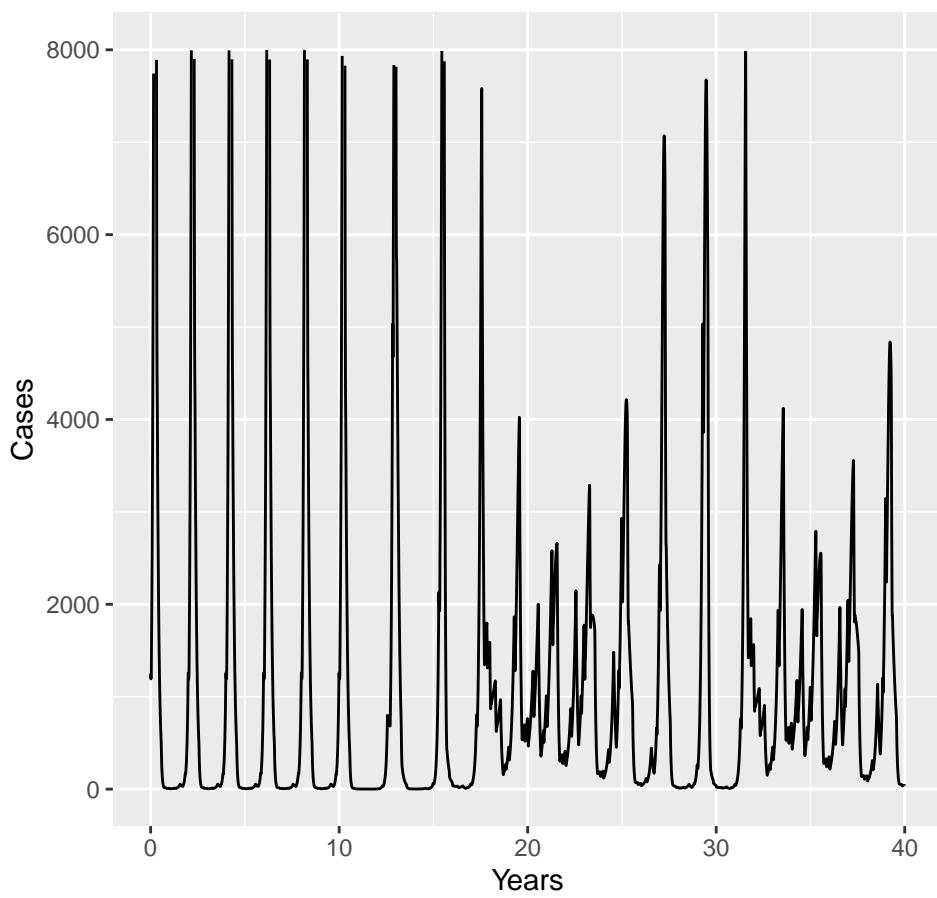


```
x<-det_SEIR_gamma(3.3e6,12,8,5,8,5,20/(1000*364),0.21,50,10,30,0.3)

# gc()

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.21, vacc = 0.3") + ylim(0,8000)
```

alpha = 0.21, vacc = 0.3

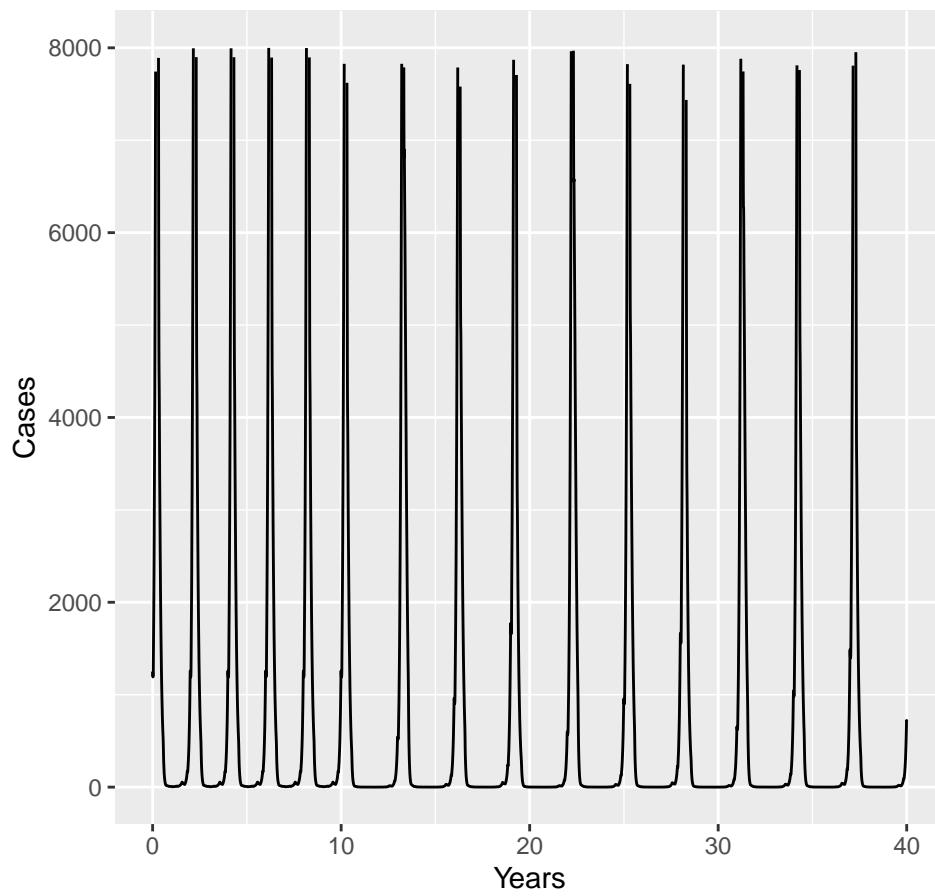


```
x<-det_SEIR_gamma(3.3e6,12,8,5,8,5,20/(1000*364),0.21,50,10,30,0.4)

# gc()

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.21, vacc = 0.4") + ylim(0,8000)
```

alpha = 0.21, vacc = 0.4

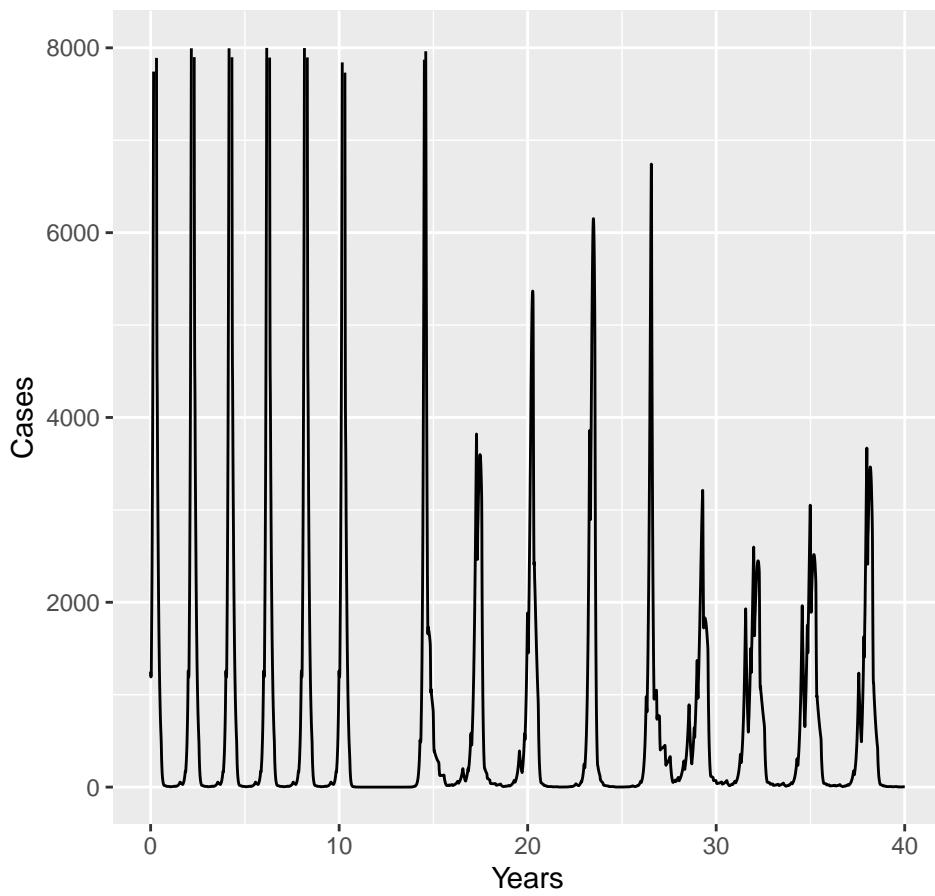


```
x<-det_SEIR_gamma(3.3e6,12,8,5,8,5,20/(1000*364),0.21,50,10,30,0.6)

# gc()

ggplot(x,aes(x=time,y=I*(1/0.4)*(5/7))) + geom_line() + xlab('Years') + ylab('Cases') +
  ggtitle("alpha = 0.21, vacc = 0.6") + ylim(0,8000)
```

alpha = 0.21, vacc = 0.6



[Return to P187](#)

Deterministic model

For the standard *SEIR* model with constant rates of progression through the exposed and infectious compartments an R_0 of ~ 4.0 gives a rough calibration to the observed data (black points and lines):

```

SEIR_gamma <- function(t,y,theta) {

  x_i = integer(2);
  beta = theta[1];
  TE = theta[2];
  TI = theta[3];
  x_i[1] = theta[4];
  x_i[2] = theta[5];

  I_tot=0;
  N_tot=0;

  dy_dt = numeric(2+x_i[1]+x_i[2]);

  for(k in (1+x_i[1]+1):(1+x_i[1]+x_i[2])){I_tot = I_tot + y[k];}
  for(k in 1:(2+x_i[1]+x_i[2])){N_tot = N_tot + y[k];}

  lambda = beta*I_tot/N_tot

  #Susceptibles
  dy_dt[1] = - (lambda) * y[1];
  #First stage of Exposed (but not yet infectious)
  dy_dt[2] = lambda * y[1] - (x_i[1]/TE) * y[2];

  #If shape parameter > 1 update internal exposed stages
  if(x_i[1]>1){
    for(k in 3:(1+x_i[1])){
      {dy_dt[k] = (x_i[1]/TE) * y[k-1] - (x_i[1]/TE) * y[k];}
    }
    #First infectious stage
    dy_dt[(1+x_i[1]+1)] = (x_i[1]/TE)*y[(1+x_i[1])] - (x_i[2]/TI)*y[(1+x_i[1]+1)];
    #If more than one infectious stage, run through internal stages
    if(x_i[2]>1){
      for(k in (3+x_i[1]):(1+x_i[1]+x_i[2])){
        {dy_dt[k] = (x_i[2]/TI) * y[k-1] - (x_i[2]/TI) * y[k];}
      }
    }
    # Final absorbing recovered stage
    dy_dt[(1+x_i[1]+x_i[2]+1)] = (x_i[2]/TI)*y[(1+x_i[1]+x_i[2])];

    return(list(dy_dt))
  }

det_SEIR <-function(R0,TE,TI,shape_E,shape_I,vacc)
{
  N0 = 763.0

  theta = numeric(5)

  beta = R0/TI
  s0 = (1-vacc)*(N0-1)
  e0 = 1
  i0 = 0
  r0 = vacc*(N0-1)

  y0 = numeric(2+shape_E+shape_I);
  y0[1] = s0;
  y0[2] = 1
  y0[2+shape_E+shape_I] = r0

  theta[1] = 1
  theta[2] = 1
  theta[3] = 1
  theta[4] = 1
  theta[5] = 1
}

```

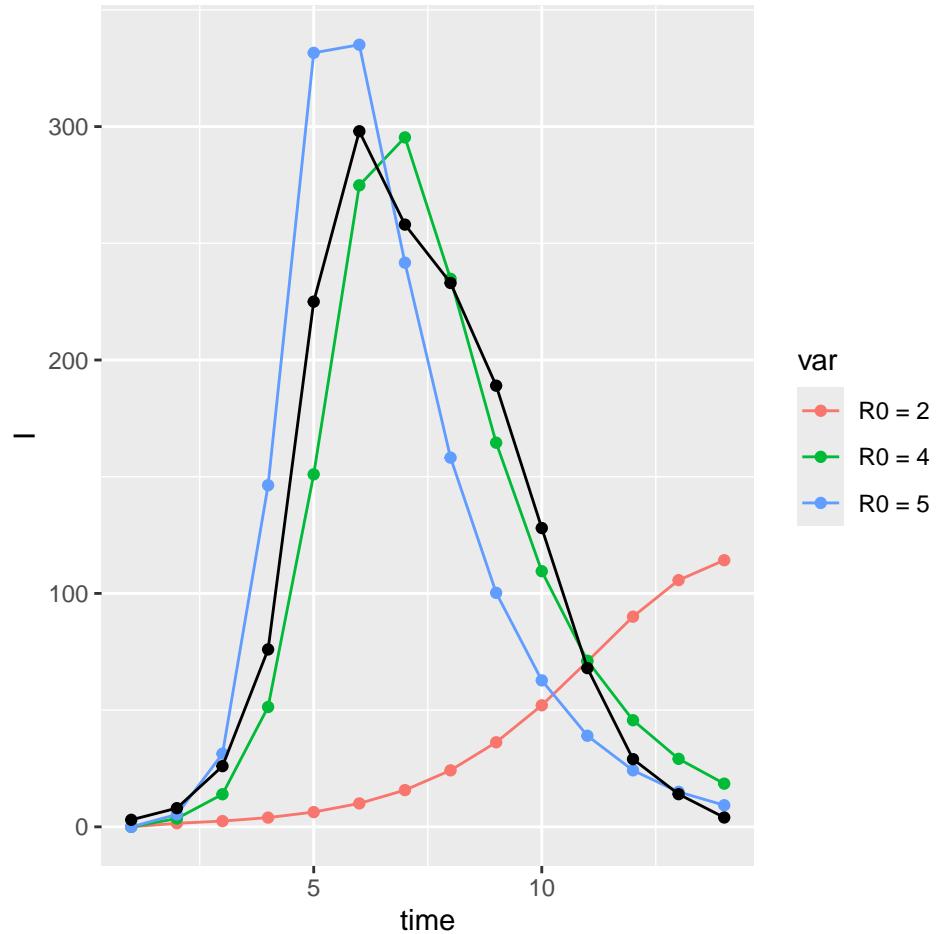
```
# Standard SEIR model (exponential latent, infectious periods)

expI <- det_SEIR(2.0,0.04,2.0,1,1,0.0)
det_calibrate <- tibble(time=1:14,var='R0 = 2',I=expI[1:14])

expI <- det_SEIR(4.0,0.04,2.0,1,1,0.0)
det_calibrate <- det_calibrate %>% bind_rows(tibble(time=1:14,var='R0 = 4',I=expI[1:14]))

expI <- det_SEIR(5.0,0.04,2.0,1,1,0.0)
det_calibrate <- det_calibrate %>% bind_rows(tibble(time=1:14,var='R0 = 5',I=expI[1:14]))

ggplot(det_calibrate,aes(x=time,y=I,col=var)) + geom_line() + geom_point() +
  annotate('point',1:14,targetI[1:14]) + annotate('line',1:14,targetI[1:14])
```



This rough calibration does not hold up if we now vary the distributional assumption for the latent and infectious periods. The supplied code has implemented the *SEIR* model with gamma distributed (strictly an Erlang distribution were the shape parameter is an integer) latent and infectious periods. If we vary the shape parameters for I and E we see that the peak prevalence and initial rate of exponential growth are very sensitive to changes in the distribution. Less dispersed distributions (i.e. higher shape parameters) have sharper epidemics with higher peak prevalence. The final size (and approximate value of R_0) on the other hand are insensitive to these changes. Below we plot a set of three curves below for shape parameters = 10 illustrating that we now require a lower value of R_0 to (roughly) match the data. The fit is not as good as I deliberately picked the average infectious and latent periods to give a reasonable fit with the exponential model...

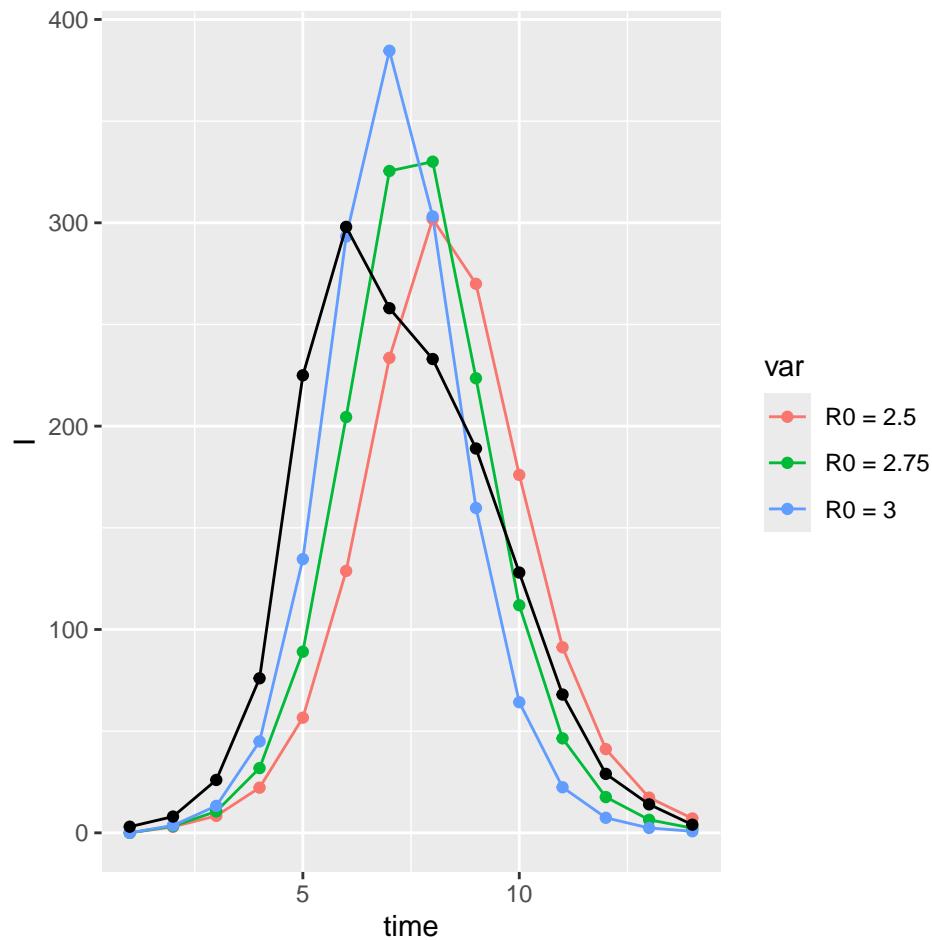
```
# Standard SEIR model (gamma 10 latent, infectious periods)

expI <- det_SEIR(2.5,0.04,2.0,10,10,0.0)
det_calibrate <- tibble(time=1:14,var='R0 = 2.5',I=expI[1:14])

expI <- det_SEIR(2.75,0.04,2.0,10,10,0.0)
det_calibrate <- det_calibrate %>% bind_rows(tibble(time=1:14,var='R0 = 2.75',I=expI[1:14]))

expI <- det_SEIR(3.0,0.04,2.0,10,10,0.0)
det_calibrate <- det_calibrate %>% bind_rows(tibble(time=1:14,var='R0 = 3',I=expI[1:14]))


ggplot(det_calibrate,aes(x=time,y=I,col=var)) + geom_line() + geom_point() +
  annotate('point',1:14,targetI[1:14]) + annotate('line',1:14,targetI[1:14])
```



In practice the latent and infectious period distributions are not identifiable purely from a single epidemic curve. As this very simple comparison demonstrates calibrating our model based on the wrong assumption can lead to a significant under or over estimate of R_0 .

[Return to P190](#)

Stochastic model

The rough calibration we identified using the deterministic model mostly holds up for the stochastic model with replicates scattering around the observed numbers. Although the population is small, the transmission rate is

relatively high ($R_0 = 4$) so the deterministic model is a reasonable approximation for the average of the stochastic simulations. If the transmission rate was lower this would not necessarily be the case.

```

SEIR_rates <- function(state,p)
{
rate <- numeric(2)
# Probability of a transmission sum(state) = population size N
rate[1] <- p['beta']*state['S']*state['I']/sum(state)
# Probability of emergence from E to I
rate[2] <- p['sigma']*state['E']
# Probability of recovery from I to R
rate[3] <- p['gamma']*state['I']
return(rate)
}

SEIR_events <- function(state,rate)
{
  total_rate <- sum(rate)
  # Draw single random variate
  x <- runif(1)
  if(x*total_rate < rate[1])
  { # Transmission
    state['S'] = state['S'] - 1
    state['E'] = state['E'] + 1
  }else if(x*total_rate < sum(rate[1:2]))
  { # Emergence
    state['E'] = state['E'] - 1
    state['I'] = state['I'] + 1
  }else{
    # Recovery
    state['I'] = state['I'] - 1
    state['R'] = state['R'] + 1
  }
  return(state)
}

stochastic_rep_SEIR <- function(R0,TE,TI,vacc,tmax)
{
n0 = 763.0
i0 = 1
# State Vector (One variable)
state <- c('S'=round((n0-1)*(1-vacc)), 'E'=0, 'I'=i0, 'R'=round(vacc*(n0-i0)))
# Parameters
p = c('beta'=R0/TE, 'sigma'=1/TE, 'gamma'=1/TI)

# Time Variable
t <- 0.0;
# Save I and R only (I to compare to data, R to calculate final size)
# Could record all states but will be more costly in terms of memory use and run time
# so omitted here.
# Subtract off initial number vaccinated so that R counts total cumulative infections
output <- tibble(t=0,I=state['I'],R=state['R']-round(vacc*(n0-i0)))

# Calculate initial rate of events
rate_vector <- SEIR_rates(state,p)
total_rate = sum(rate_vector)
# Main Loop
repeat
{
  # Update time - sample time increment from exponential distribution
  # with rate given by total rate
  t = t + rexp(1,total_rate)
  # Simulate the event using relative rates in rate_Vector
  state <- SEIR_events(state,rate_vector)
  # Output
}

```

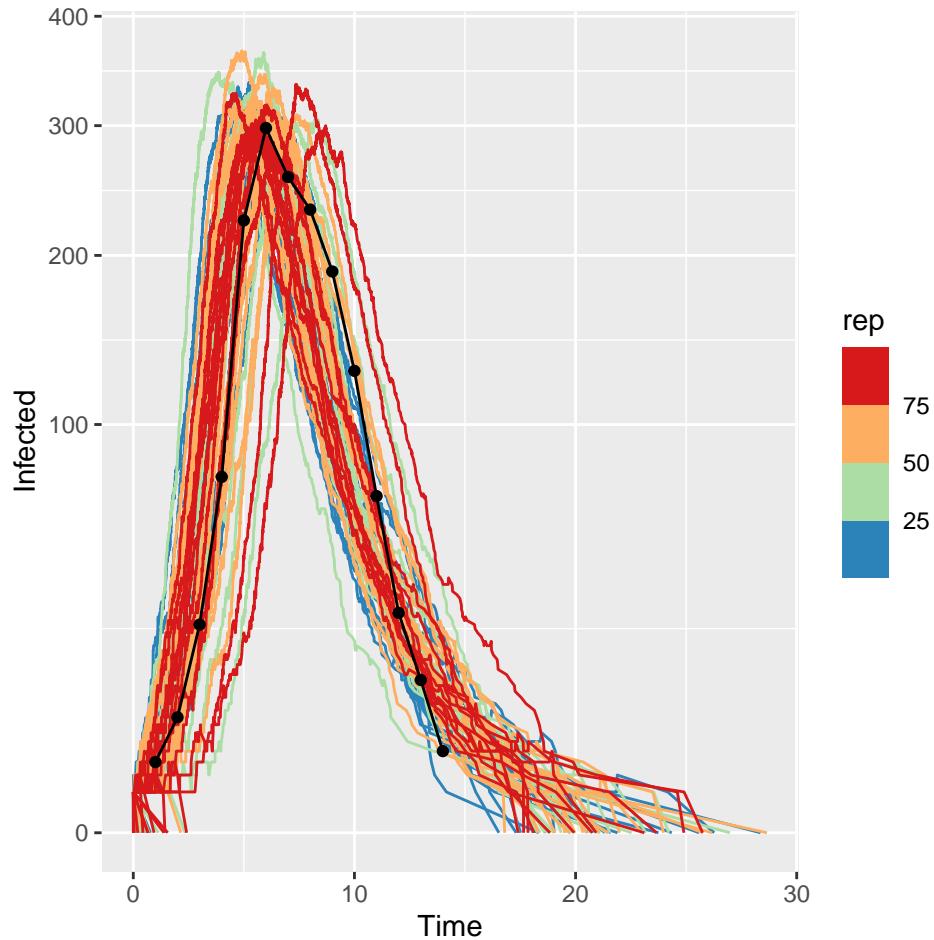
```

out <- tibble()

for(i in 1:100) {
  out <- out %>% bind_rows(stochastic_rep_SEIR(4.0,0.04,2.0, 0.0, 30) %>% mutate(rep=i))
}

ggplot(out,aes(x=t,y=I,col=rep,group=rep)) + geom_path() + scale_y_sqrt() +
  annotate('point',1:14,targetI[1:14]) + annotate('line',1:14,targetI[1:14]) +
  scale_color_fermenter(palette = "Spectral") + xlab('Time') + ylab('Infected')

```



```

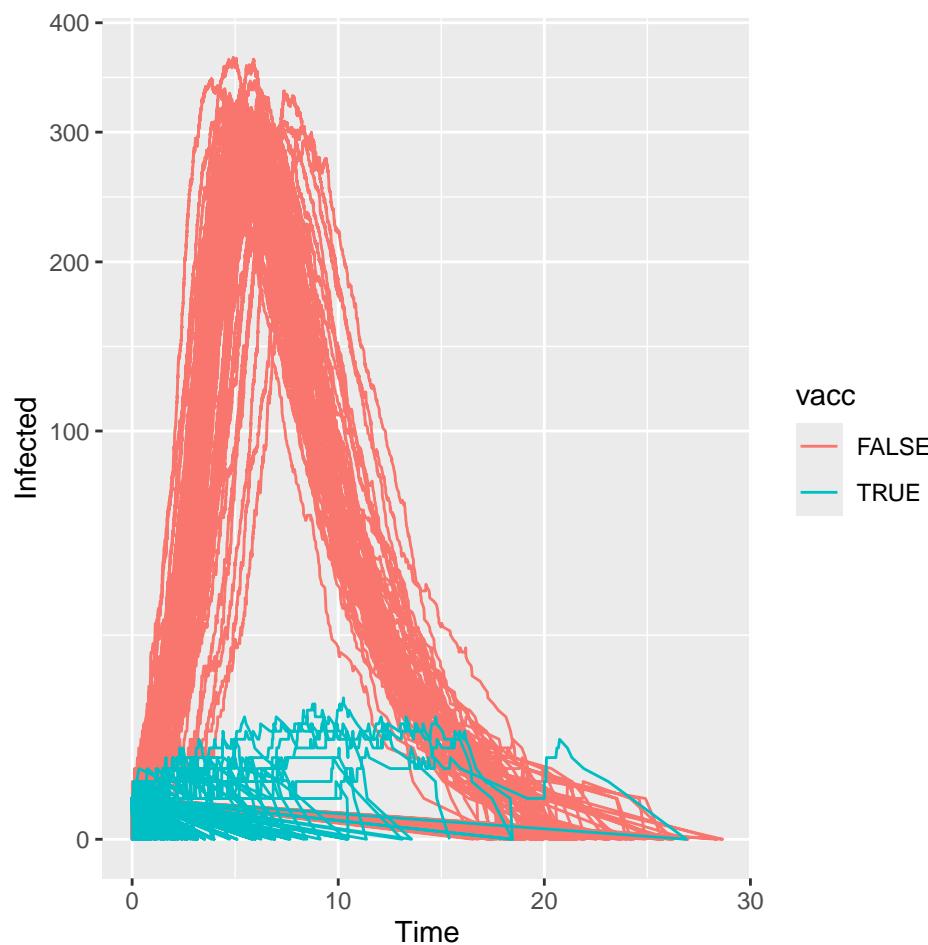
outvacc <- tibble()

for(i in 1:100) {
  outvacc <- outvacc %>% bind_rows(stochastic_rep_SEIR(4.0,0.04,2.0, 0.8, 30) %>% mutate(rep=i))
}

outagg <- (out %>% mutate(vacc=FALSE)) %>% bind_rows(outvacc %>% mutate(vacc=TRUE))

ggplot(outagg,
       aes(x=t,y=I,col=vacc)) + geom_path() + xlab('Time') + ylab('Infected') + scale_y_sqrt()

```

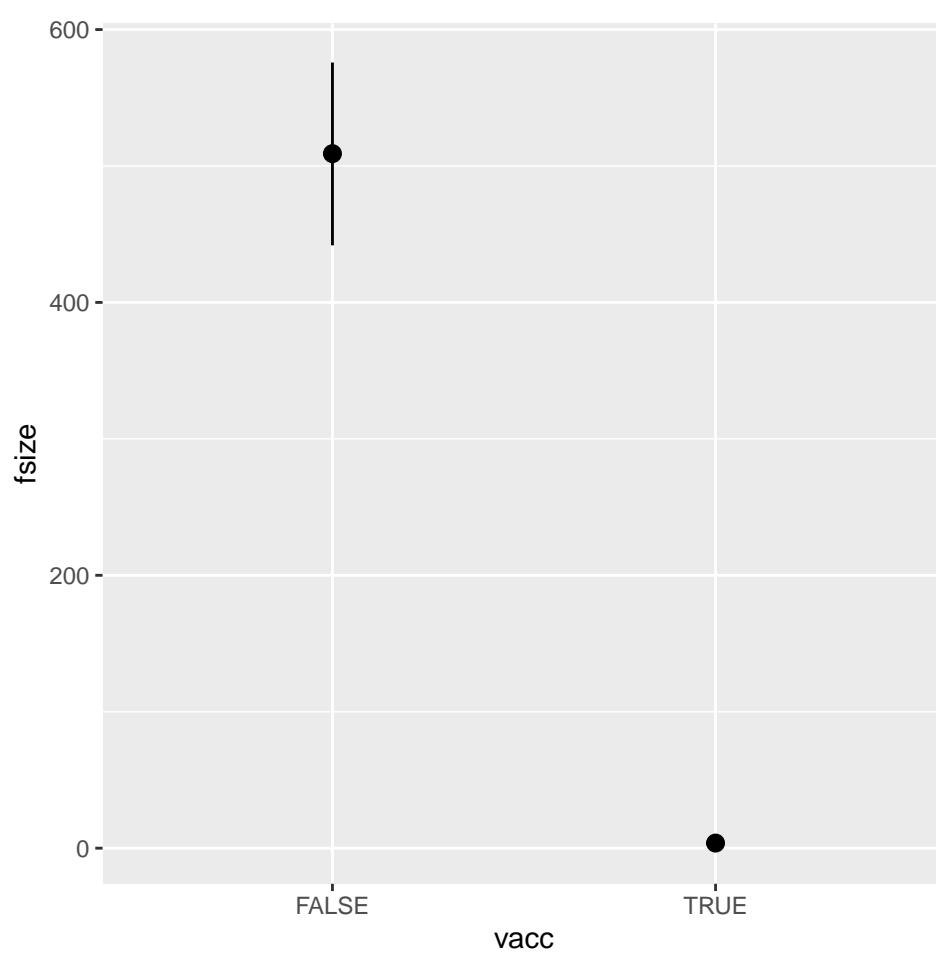


The main reason for considering a stochastic model for this question is then to quantify how vaccination changes the chances of seeing an outbreak at all. We can calculate this from our simulated scenarios by calculate the mean number of simulations that have more than one infection (`mean(outbreak)` in table below) or compare the final size distribution for the two scenarios.

```
outagg %>%
  group_by(rep,vacc) %>%
  summarise(outbreak=max(R)>1) %>%
  ungroup() %>%
  group_by(vacc) %>%
  summarise(mean(outbreak))
```

```
## # A tibble: 2 x 2
##   vacc   `mean(outbreak)`
##   <lgl>     <dbl>
## 1 FALSE      0.75
## 2 TRUE       0.36
```

```
ggplot(outagg %>%
        group_by(rep,vacc) %>%
        summarise(fsize=max(R)),
        aes(x=vacc,y=fsize)) +
  stat_summary(fun.data = "mean_cl_boot")
```



[Return to P190](#)