Spatial microsimulation with R

Contents

1 Introduction

Spatial microsimulation is statistical technique for combining individual-level datasets with geographical data and analysing the resulting *spatial microdata*. The term is little known outside the fields of human geography and regional science. Yet the underlying methods have the potential to be useful in a wide range of applications. Spatial microsimulation, as taught in this book, can be of use to local housing administrators, transport planners and researchers hammering out the details of how society could operate in a post carbon world — after we stop burning fossil fuels.

There is growing interest in spatial microsimulation. This is due largely to its practical utility in an era of 'evidence-based policy' but is also driven by changes in the wider research environment inside and outside of academia. Continuous improvements in computers, advances in the availability and capabilities of software for handling data, and the 'open data' movement all mean it is now easier than ever to simulate the populations of small administrative areas at the individual-level, almost anywhere in the world.

Still, the term is shrouded by an unnecessary mystery. This is partly because the technique is inherently difficult to understand and partly, I argue, due to researchers themselves. Some of the literature that uses the term is ambiguous, inconsistent, not reproducible and baffling for the newcomer. Some writing on the subject adds to the confusion by treating spatial microsimulation as a magical black box or by not defining terms.

In fact, spatial microsimulation means different things to different people. In the social sciences, spatial microsimulation has two main meanings, as a technique or an approach: 1. A *technique* for generating spatial microdata — individuals allocated to zones. 2. An *approach* to modelling based on spatial microdata, simulated or real.

Throughout this book we will see spatial microsimulation as both technique and as a broader approach, generally moving from the former to the latter perspective as the chapters progress.

Another issue tackled in this book is reproducibility. Most findings in the reports and papers in the field cannot easily be replicated by their readers. In today's age of fast Internet connections, open access datasets and free software, there is little excuse for this. Non-reproducibility is damaging: if your methods are hidden, how can others benefit from them? If your analysis cannot be independently verified, how can others take the results seriously? They cannot. This book encourages good practice in reproducibility[1] by providing the tools for its readers to actually *do* spatial microsimulation, on realistic data for plausible purposes.

To understand the importance of reproducibility, consider two more questions. If a small community of researchers hold the majority of the know-how and computer code needed to perform spatial microsimulation, how can it grow and spread to other applications? If the results of most spatial microsimulation research are not reproducible, how can policy makers and society at large be expected to trust them?

This final point is critical not only to spatial microsimulation but the disciplines to which it contributes. Reproducibility is a prerequisite of falsifiability and falsifiability is the backbone of science (Popper, 1959). By producing non-reproducible research, researchers risk damaging not only their own integrity, but the scientific credibility of the disciplines in which they work. These philosophical antecedents inform the book's practical nature. The aim is simple: to provide an accessible yet deep foundation in spatial microsimulation. This involves both *understanding* and *implementation* of the technique. From research into 'experiential learning', it has been found that doing something is often the best way towards understanding.

This book demonstrates that spatial microsimulation need not be an abstract process that one simply reads about. It is a living, evolving set of techniques, methods and code, not a prescriptive formula for arriving at the 'right' answer. In other words, spatial microsimulation is an art, practiced by a growing number of people worldwide. It is hoped that this book contributes to the community that uses spatial microsimulation and that it encourages collaboration, innovation and rigour. Above all, the book aims to make spatial microsimulation accessible, a practical tool used by anyone with the know-how. As Kabakoff (2011 p. xxii) put it regarding R, "the best way to learn is to experiment". The same applies to spatial microsimulation.

[1] TODO: add table from Peng (2006) on the criteria for reproducible research

1.1 Why spatial microsimulation with R?

```
# expressing oneself.^[This video introduces the idea of
# expressing oneself in [R](http://youtu.be/wkiOBqlztCo)].
```

Software decisions have a major impact on flexibility, efficiency, reproducibility and ease of expressing oneself. Nearly 3 decades ago Hölm (1987, p. 153) observed that "little attention is paid to the choice of programming language used" for microsimulation. This appears to be as true now as it was then: software is rarely discussed in spatial microsimulation papers. Yet the decision of which software to use will have a lasting impact on your research, what you can and cannot do with your code and opportunities for collaboration.

In my own research, for example, a conscious decision was made early on to use R. This had subsequent knock-on impacts on the features, analysis and even design of my simulations. There are hundreds computer programming languages and many of these are general purpose and 'Turing complete', meaning they could, with sufficient effort, perform spatial microsimulation (or any other numerical operation). So why choose R?

There are many factors that should influence the selection of a suitable language, the most important of which include flexibility, speed of processing and, most importantly for research, ease and speed of writing, adapting and communicating code. R excels in each of these areas (with the possible exception of speed, a problem that can be overcome by judicious use of add-on packages to the base R installation).

R is a high level language for statistical computing compared with general purpose languages such as C and Python. This means that instead of having to write code to perform statistical operations from scratch, a pre-made function probably already exists. To calculate the mean value of any variable `x` in pure Python, for example, one would need to type 20 characters: `float(sum(x))/len(x)`.[2]

In pure R 7 characters are sufficient: `mean(x)`. One may argue that saving a few keystrokes while writing code is not a priority, and this may be true, but it is certain that the time savings of being concise can be vast. The example of calculating the mean in R and Python illustrates a wider point: R was *designed* to work with statistical data, so many functions in the default R installation (e.g. `lm()`, to create a linear regression model) perform statistical analysis. In agent-based modelling the statistical analysis of results often occupies more researcher time than running the model itself (Theile and Grimm, 2012); Theile, 2014). The same applies to spatial microsimulation, necesitating statistical software such as R.

R has an active and growing user community. As a result there are thousands of packages that extend R's capabilities by providing new functions to the user. The **ipfp** package, for example, can greatly reduce the computational time taken for a key element of spatial microsimulation process, as we shall see in ipfp section.

1.2 Learning the R language

Having learned a little about *why* R is a good tool for the job, it is time to think about *how* R should be used. The most useful high-level advice I received on the subject is to think of R not as a series of isolated commands, but as an interconnected *language*, in the fullest sense of the word, like Spanish: frequent practice, persistence and experimentation are needed for deep learning.

Indeed, R can be seen as two separate components: the *interpreter* is what sends instructions to the computers processor. The R *language* provides a way of expressing oneself and explaining ideas to other human beings. Critical to its role as a language is R's unique *syntax*, which allows relatively complex numerical ideas to be described with a small number of keystrokes.

[2] The `float` function is needed in case whole numbers are used. This can be reduced to 13 characters with the excellent **NumPy** package: `import numpy; x = [1,3,9]; numpy.mean(x)` would generate the desired result. The R equivalent is `x = c(1,3,9); mean(x)`.

1.3 Typographic conventions

The following typographic conventions are followed to make the practical examples easier to follow:

- In-line code is povided in `monospace` font to show it's something the computer understands.
- Larger blocks of codes, referred to as *listings*, are provided on separate lines and have coloured *syntax highlighting* to distingish between values, names and functions:

```
x <- c(1, 2, 5, 10) # create a vector
sqrt(x) # find the square route
```

```
## [1] 1.000 1.414 2.236 3.162
```

- Output from the *R console* is preceded by the `##` symbol, as illustrated above.
- Comments are preceded by a single `#` symbol to explain specific lines.

There are many ways to write R code that will generate the same results. However, to ensure clarity and consistency the style guide in Hadley Wickham's *Advanced R* book (Wickham, 2014) is followed throughout. Consistent style and plentiful comments will ensure code readable by yourself and others for decades to come.

1.4 An overview of the coursebook

This coursebook builds on the tutorial *Introducing spatial microsimulation with R: a practical* (Lovelace, 2014) with improved code and explanation. The booklet is a precurser to a book CRC Press's R Series which will be published in summer 2015. Therefore any comments on the code, explanation or contents will be gratefully received.[3]

The structure is as follows:

[3] Feedback can be left via email to r.lovelace@leeds.ac.uk or via the project's GitHub page.

2 What is spatial microsimulation?

Spatial microsimulation, as used in this book, can be defined as a statistical technique for allocating individuals from a survey dataset to administrative zones, based on shared variables between the areal and individual level data.

However, as with many new and infrequently used phrases, this understanding is not shared by everyone. The meaning of spatial microsimulation varies depending on context and who you ask: to an economist, for example, spatial microsimulation is likely to imply modelling some kind of temporal process such as how individual agents in different areas respond to changes in prices or policies. To a transport planner, the term may imply simulating the precise movements of vehicles on the transport network. To your next door neighbour it may mean you have started speaking gobbledygook! Hence the need to define our terminology.

2.1 Terminology

Delving a little into the etymology and history of the term reveals the reasons behind this duplicity of meaning and highlights the importance of terminology. Only in very few contexts will one be understood when one says "I use *spatial microsimulation*" in everyday life. Usually it is important to add context. Below are a few hypothetical situations and suggestions of how one could respond to them.

- When talking to a colleague, a transport modeller: "spatial microsimulation, also known as population synthesis..."
- Speaking to agent based modellers: "we use spatial microsimulation to simulate the characteristics of geo-referenced agents..."
- Communicating with undergraduates who are unlikely to have come across the term or its analogies. "I do spatial microsimulation, a way of generating individual-level data for small areas..."
- Chatting casually in the pub or coffee shop: "I'm using a technique called spatial microsimulation to model people...".

The above examples illustrate that there is great potential for confusion and shows that care needs to be taken to tailor the words used depending on the target audience. All this links back to the importance of transparency and reproducibility of method discussed in .

Faced with uncomprehending stares when describing the method, some may be tempted to 'blind them with science', relying on sophisticated-sounding jargon, for example by saying: "we use simulated annealing in our integerised spatial microsimulation model". Such wording obscures meaning (how many people in the room will understand 'integerised', let alone 'simulated annealing') and makes the process sound inaccessible. Although much jargon is used in the spatial microsimulation literature in this book, care must be taken to ensure that people understand what you are saying.

This raises the question, why use the term spatial microsimulation at all, if it is understood by so few people? The answer to this is that spatial microsimulation, defined clearly at the outset and used correctly, can usefully describe a technique that would otherwise need many more words on each use. Try replacing 'spatial microsimulation' with 'a statistical technique to allocate individuals from a survey dataset to administrative zones based on shared variables between the areal and individual level data' each time it appears in this book and the advantages of a simple term should become clear. 'Population synthesis' is perhaps a more accurate term but, transport modelling aside, the literature already uses 'spatial microsimulation'. Rather than create more complexity with *another* piece of jargon, it is best to continue with the term favoured by the the majority of practitioners.

Why has this situation, in which practitioners of a statistical method must tread carefully to avoid confusing their audience, come about? First it's worth stating that the problem is by no means unique to this field:

imagine the difficulties that Bayesian statisticians must encounter when speaking of prior and posterior probability distributions to an uninitiated audience. Let alone describing Gibb's sampling. To more precisely answer the question, and gain an insight into the origins of the definition provided at the outset of this chapter, we consider the beginnings and evolution of the term in written work.

2.2 The etymology of spatial microsimulation

2.3 Spatial microsimulation as an approach to modelling

2.4 What spatial microsimulation is not

Spatial microsimulation is not strictly spatial

The most surprising feature of spatial microsimulation, as used in the literature, is that *the method is not trictly* spatial*. The only reason why the methodology has developed this name (as opposed to 'small area population synthesis', for example) is that practitioners tend to use administrative zones, which represent geographical areas, as the grouping variable. However, any mutually exclusive grouping variable, such as age band or number of bedrooms in your house, could be used. Likewise, geographical location can be used as a* constraint variable*. In most spatial microsimulation models,* the spatial variable is a mutually exclusive grouping, interchangeable with any such group*. "Spatial" is thus 1st on the list of things that spatial microsimulation is not.

To be more precise, spatial microsimulation is not *inherently spatial*. Spatial attributes such as the geographic coordinates of home and work locations can easily be added to the spatial microdata after they have been generated, and the use of geographical variables as the grouping variable is critical here.

Spatial microsimulation is not agent-based modelling (ABM).

2.5 An illustrative example from SimpleWorld

To see the link between the abstract methology introduced later in the book — in basics-chapter — and the various real-world applications explored in this chapter, let's take a look at a simple example of the kind of situation where spatial microsimulation is needed. This book is primarily methodological, so hopefully this example will help bridge the gap between method and application and whet the reader's appetite for learning how to perform spatial microsimulation using R.

We'll use an imaginary world called SimpleWorld, consiting of only 3 zones that cover the entirety of the SimpleWorld sphere (exluding the poles; see Figure xx below).

This is a small world, containing 12, 10 and 11 individuals of its alien inhabitants in each zone, 1 to 3, respectively. From the SimpleWorld Census, we know how many young (under 49 space years old) and old (over 50) residents live in each zone, as well their genders: male and female. This information is displayed in the tables below, and is plotted on the Mercator maps below.

zone	0-49 yrs	50 + yrs
1	8	4
2	2	8
3	7	4

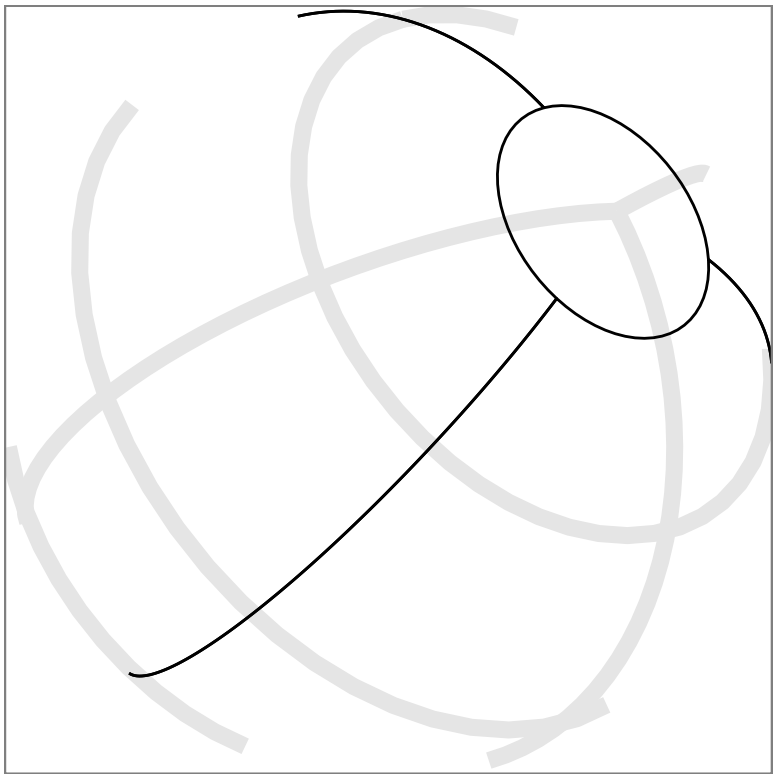
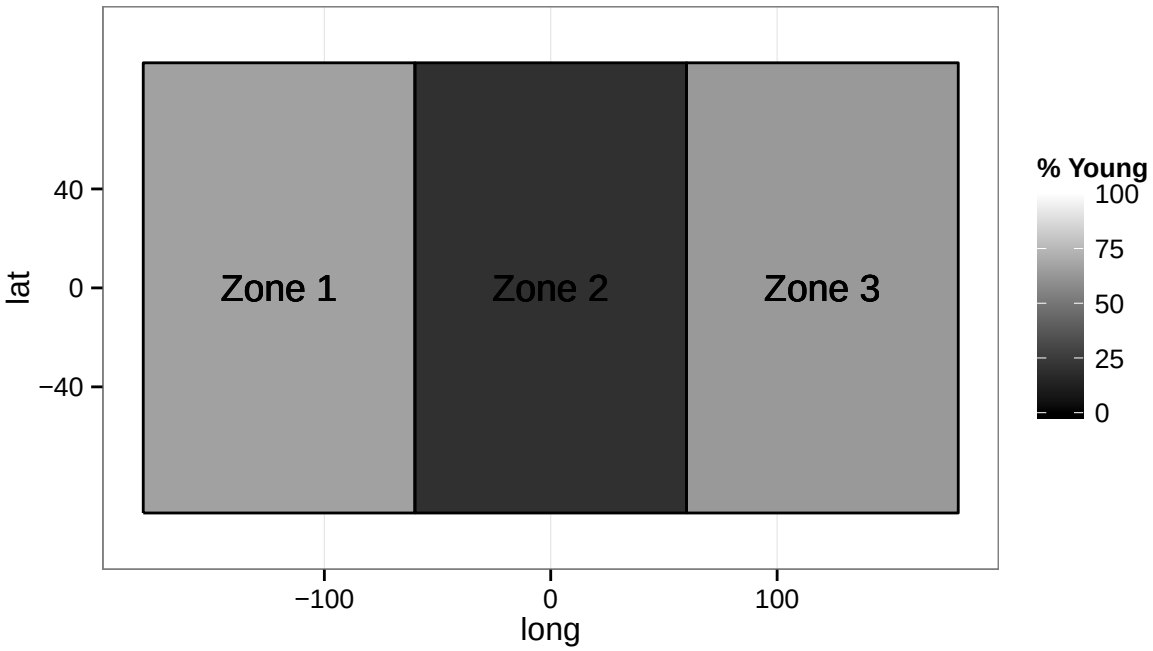
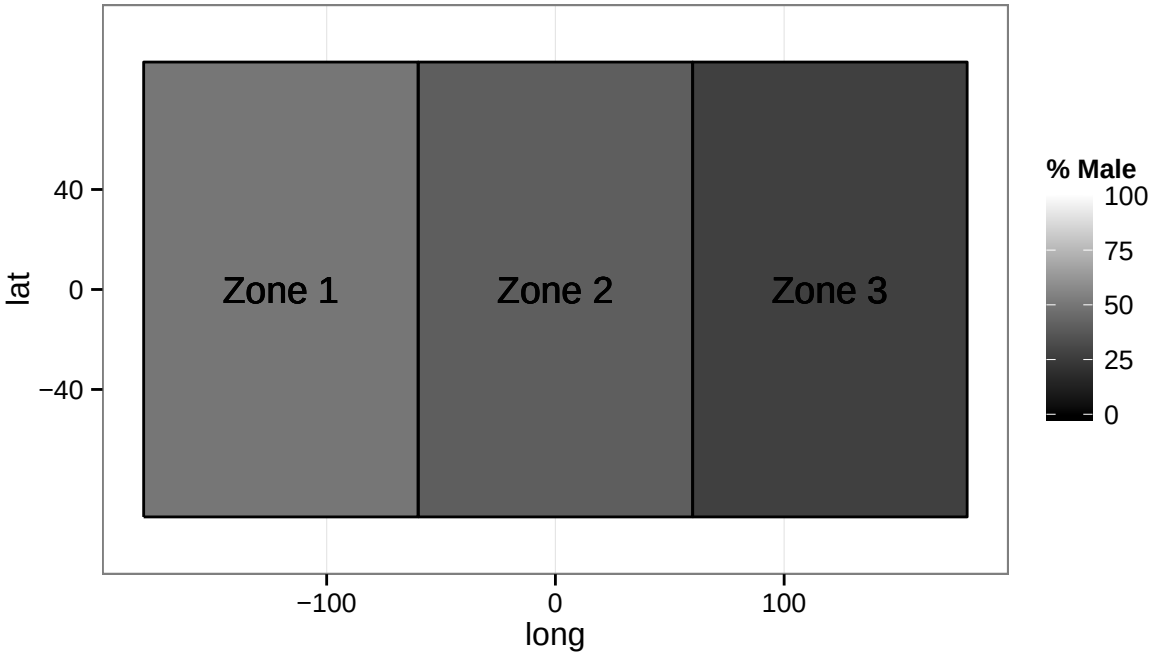


Figure 1: plot of chunk SimpleWorld-globe

Zone	m	f
1	6	6
2	4	6
3	3	8





Next imagine a more detailed dataset about 5 of SmallWorlds' inhabitants, recorded from a survey. This is in a different form from the aggregate-level data presented in the above tables. This *microdata* survey contains one row per individual, in contrast to the *aggregate constraints*, which have one row per zone. This individual level data includes exact age (not just the crude and unflattering categories of "young" and "old"), as well as income:

id	age	sex	income
1	59	m	2868
2	54	m	2474
3	35	m	2231
4	73	f	3152
5	49	f	2473

Note that although the microdataset contains additional information about the inhabitants of SmallWorld, it lacks geographical information about where each inhabitant lives or even which zone they are from. Spatial microsimulation tackles this problem, by allocating individuals from a non-geographical dataset to geographical zones in another.

The procedures we will learn to use in this book do this by allocating *weights* to each individual for each zone. The higher the weight for a particular individual-zone combination, the more representative that individual is of that zone. This information can be represented as a *weight matrix*, such as the one shown below.

Individual	Zone 1	Zone 2	Zone 3
1	1.228	1.725	0.725
2	1.228	1.725	0.725
3	3.544	0.550	1.550
4	1.544	4.550	2.550
5	4.456	1.450	5.450

The highest value (5.450) is located, to use R's notation, in cell [5,3], the 5th row and 3rd column in the matrix. This means that individual number 5 is considered to be highly representative of Zone 3, given the input data in SimpleWorld. This makes sense because there are many (7) young people and many (8) females in Zone 3, relative to the input microdataset (which contains only 1 young female). The lowest value (0.550) is found in cell [3,2]. Again this makes sense: individual 3 from the microdataset is a young male yet there are only 2 young people and 4 males in zone 2. A special feature of the weight matrix above is that each of the column sums is equal to the total population in each zone. We will discover how the weight matrices are generated in chapter4.

Another output from spatial microsimulation, that can be generated from the above weight matrix or by other approaches is *spatial microdataset*. This is dataset that contains a single row per individual (as with the input microdata) but also an additional variable indicating where each individual lives. The challenge is to ensure that the spatial microdataset is as representative as possible of the aggregate constraints, while only sampling from a realistic baseline population. A feasible combination of individuals sampled from the microdata that represent zone 2 is presented in table xx below; the complete spatial microdataset allocates whole individuals to each zone, resulting in a more or less realistic insight into the inhabitants of SimpleWorld and for the purposes of modelling.

id	age	sex	zone
1	59	m	2
2	54	m	2
4	73	f	2
4	73	f	2
4	73	f	2
4	73	f	2
5	49	f	2
1	59	m	2
4	73	f	2
5	49	f	2

The table is a reasonable approximation of the inhabitants of zone 1: older females dominate in both the aggregate (which contains 8 older people and 6 females) and the simulated spatial microdata (which contains 8 older people and 7 females). We will learn how to create such *integerised* datasets during the course of this book.

But how are these outputs *useful*?

The above example should also flag in the reader's mind some limitations of the methodology advocated in this book: spatial microsimulation will only yield useful results if the input microdataset is representative

of the population as a whole, and for each region. If the relationship between age sex is markedly different in one zone compared with what we assume to be the global averages of the input data, for example, our estimates could be way-out. Using such a small sample, one could rightly argue, how could the diversity of 33 inhabitants of SimpleWorld be represented by our simulated spatial microdata? This question is equally applicable to larger simulations. These issues are important and will be tackled in section validation.

```
# Applications of spatial microsimulation - to be completed!

## Updating cross-tabulated census data

## Economic forecasting

## Small area estimation

## Transport modelling

## Dynamic spatial microsimulation

## An input into agent based models
```

3 Spatial microsimulation with R: the basics

In this chapter we progress from an understanding of the theory and applications actually undertake a spatial microsimulation. The example is small and simple, but the same principles used here will apply to larger and more complex projects. It is worth spending time to master these basics: subsequent steps will be much easier.

The aim of this chapter is to provide a deep insight into how to use R for spatial microsimulation. The easiest way to access the data used in this chapter (and the data for all other chapters), the easiest way is to download and unzip the book's GitHub repository. From there, you will want to run R from the project's root directory.

3.1 The input data

As with most spatial microsimulation models, this example consists of a non-geographical individual-level dataset and a series of geographical zones. This section focuses on the data: loading, manipulating and assessing the R objects that will eventually be used as inputs in the spatial microsimulation model run.

To ease reproducibility of the analysis, it is recommended that the process begins with a copy of the *raw* input dataset on one's hard disc. Rather than modifying this file, modified ('cleaned') versions should be saved as separate files. This ensures that after any mistakes, one can always recover information that otherwise could have been lost and makes the project fully reproducible.

It sounds trivial, but the *precise* origin of the input data should be described. Comments in code that loads the data (and resulting publications), allows you or others to recall the raw information.

The process of loading, checking and preparing the input datasets for spatial microsimulation is generally a linear process, encapsulating the following stages:

1. Load orginal data
2. Remove excess information
3. Re-categorise individual-level data
4. 'Flatten' individual-level data
5. Set variable and value names

'Stripping down' the datasets so that they only contain the bare essential information will enable you to focus solely on the data that you are interested in. This is not covered in this chapter because the input datasets are already extremely bare and because the process should be obvious.

We start with the individual-level dataset for a reason: this dataset is often more problematic to format than the constraint variables, so it is worth becoming acquainted with it at the outset. Of course, it is possible that the data you have are not suitable for spatial microsimulation because they lack sufficient constraint variables with shared categories in both individual and aggregate level tables. We assume that you have already checked this. The checking process for the datasets used in this chapter is simple: both aggregate and individual-level tables contain age and sex, so they can by combined. Let us proceed to load some data saved on our hard disc into R's *environment*, where it is available in RAM.

3.2 Loading and subsetting the individual-level data

The individual-level data is simply a text file in this case, that can be loaded by the following command.

```
# Load the individual-level data
ind <- read.csv("data/SimpleWorld/ind.csv")
ind # print the individual-level data
```

```
##   id age sex
## 1  1  59   m
## 2  2  54   m
## 3  3  35   m
## 4  4  73   f
## 5  5  49   f
```

In the above code a `data.frame` was loaded and assigned to object called `ind`. As shown when it is called, `ind` consists of 5 rows, each of which represents a single individual. The 3 column variables are: a unique person identification number (`id`), `age` and the gender of each individual, represented by `sex`. It is worth stage it is worth exploring the data a little more, using R's powerful and concise square bracket (`[...]`) notation: see the Appendix on subsetting for more on this.

Although `ind` is small and simple, all individual-level datasets tend to have a similar structure, it will behave in the same way as a much larger dataset. It is common, for example, to take a subset of an individual-level dataset that corresponds to the *population base* of the aggregate constraints, for example all individuals between the ages of 16 and 74 *and* in full-time education, so it is important to have a strong understanding of subsetting in R before proceeding.

3.2.1 Loading and checking aggregate-level data

Constraint data are usually made available one variable at a time, so these are read in one file at a time:

```
con_age <- read.csv("data/SimpleWorld/age.csv")
con_sex <- read.csv("data/SimpleWorld/sex.csv")
```

We have loaded the aggregate constraints. As with the individual level data, is worth inspecting each object to ensure that they make sense before continuing. Taking a look at `age_con`, we can see that this data set consists of 2 variables for 3 zones:

```
con_age
```

```
##   a0.49 a.50.
## 1     8     4
## 2     2     8
## 3     7     4
```

This tells us that there 12, 10 and 11 individuals in zones 1, 2 and 3, respectively, with different proportions of young and old people. Zone 2, for example, is heavily dominated by older people: there are 8 people over 50 whilst there are only 2 young people (under 49) in the zone.

Even at this stage there is a potential for errors to be introduced. A classic mistake with areal data is that the order in which zones are loaded changes from one table to the next. The constraint data should come with some kind of *zone id*, an identifying code that will eventually allow the attribute data to be combined with polygon shapes in GIS software.

If we're sure that the row numbers match between the age and sex tables (we are sure in this case), the next important test is to check that the total populations are equal for both sets of variables. Ideally both the *total* study area populations and *row totals* should match. If the *row totals* match, this is a very good sign that not only confirms that the zones are listed in the same order, but also that each variable is sampling from the same *population base* These tests are conducted in the following lines of code:

```
sum(con_age)
```

```
## [1] 33
```

```
sum(con_sex)
```

```
## [1] 33
```

```
rowSums(con_age)
```

```
## [1] 12 10 11
```

```
rowSums(con_sex)
```

```
## [1] 12 10 11
```

```
rowSums(con_age) == rowSums(con_sex)
```

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

3.3 Re-cateorising individual-level variables

Before transforming the individual-level dataset `ind` into a form that can be compared with the aggregate-level constraints, we must ensure that each dataset contains the same information. It is more challenging to re-categorise individual-level variables than to re-name or combine aggregate-level variables, so the former should usually be set first. An obvious difference between the individual and aggregate versions of the `age` variable is that the former is of type `integer` whereas the latter is composed of discrete bins: 0 to 49 and 50+. We can categories the variable into these bins using `cut()`:[4]

```
# Test binning the age variable
cut(ind$age, breaks = c(0, 49, 120))
```

```
## [1] (49,120] (49,120] (0,49]   (49,120] (0,49]
## Levels: (0,49] (49,120]
```

If we wanted to change these category labels to something more readable, we can do this by adding another argument to the `cut` function:

```
# Convert age into a categorical variable with user-chosen labels
(ind$age <- cut(ind$age, breaks = c(0, 49, 120), labels = c("a0_49", "a50+")))
```

```
## [1] a50+  a50+  a0_49 a50+  a0_49
## Levels: a0_49 a50+
```

Users should be ware that `cut` results in a vector of class *factor*, which can cause problems later down the line.

[4] The combination of curved and square brackets in the output may seem strange but this is in fact an International Standard - see wikipedia.org/wiki/ISO_31-11 for more information.

```
names(cons)
```

```
## [1] "a0.49" "a.50." "m"     "f"
```

3.4 Matching individual and aggregate level data names

Before combining the newly recategorised individual-level data with the aggregate constraints, it is useful to for the category labels to match up. This may seem trivial, but will save time in the long run. Here is the problem:

```
levels(ind$age)
```

```
## [1] "a0_49" "a50+"
```

```
names(con_age)
```

```
## [1] "a0.49" "a.50."
```

Note that the names are subtly different. To solve this issue, we can simply change the names of the constraint variable, assuming they are in the correct order:

```
names(con_age) <- levels(ind$age) # rename aggregate variables
```

With both the age and sex constraint variable names now matching the category labels of the individual-level data, we can proceed to create a single constraint object we label `cons`. We do this with `cbind()`:

```
cons <- cbind(con_age, con_sex)
cons[1:2, ] # display the constraints for the first two zones
```

```
##   a0_49 a50+ m f
## 1     8    4 6 6
## 2     2    8 4 6
```

3.5 'Flattening' the individual level data

We have made steps towards combining the individual and aggregate datasets and now only need to deal with 2 objects (`ind` and `cons`) which now share category and variable names. However, these datasets cannot possibly be compared because they are of different dimensions:

```
dim(ind)
```

```
## [1] 5 3
```

```
dim(cons)
```

```
## [1] 3 4
```

The above code confirms this: we have one individual-level dataset comprising 5 individuals and 3 variables (2 of which are constraint variables) and one aggregate-level constraint table comprising 6 zones for which we have counts for 4 categories across 2 variables. Clearly we need to change the dimensions of at least one object before they can be quantitatively compared. To do this we 'flatten' the individual-level dataset - meaning that we increase its width and reduce its height (number of rows) to one. This is a two-stage process. First, `model.matrix()` is used to expand each variable into the number of columns as there are categories in each. Second, `colSums()` is used to take the sum of each column.[5]

```
cat_age <- model.matrix(~ ind$age - 1)
cat_sex <- model.matrix(~ ind$sex - 1)[, c(2, 1)]
(ind_cat <- cbind(cat_age, cat_sex)) # combine flat representations of the data
```

```
##   ind$agea0_49 ind$agea50+ ind$sexm ind$sexf
## 1            0           1        1        0
## 2            0           1        1        0
## 3            1           0        1        0
## 4            0           1        0        1
## 5            1           0        0        1
```

Note that second call to `model.matrix` is suffixed with `[, c(2, 1)]`. This is to swap the order of the columns: the column variables are produced from `model.matrix` is alphabetic, whereas the order in which the variables have been saved in the constraints object `cons` is `male` then `female`. Such subtleties can be hard to notice yet completely change one's results so be warned: the output from `model.matrix` will not always be compatible with the constraint variables.

To check that the code worked properly, let's count the number of individuals represented in the new `ind_cat` variable, using `colSums`:

```
colSums(ind_cat) # view the aggregated version of ind
```

```
## ind$agea0_49  ind$agea50+     ind$sexm     ind$sexf
##            2            3            3            2
```

```
ind_agg <- colSums(ind_cat) # save the result
```

The sum of both age and sex variables is 5 (the total number of individuals): it worked! Looking at `ind_agg`, it is also clear that it has the same dimension as each row in `cons`, the aggregate-level data. We can check this by inspecting each object (e.g. via `View (ind_agg)`), although a more rigorous test is to see if `ind_agg` can be combined with `ind_agg`, using `rbind`:

```
rbind(cons[1,], ind_agg)
```

```
##   a0_49 a50+ m f
## 1     8    4 6 6
## 2     2    3 3 2
```

If no error message is displayed, the answer is yes. This shows us a direct comparison between the number of people in each category of the constraint variables in zone and and in the individual level dataset overall. Clearly, the fit is not very good, with only 5 individuals in total existing in `ind_agg` (the total for each

[5] As we shall see in xxx, only the former of these is needed if we use the **ipfp** package for re-weighting the data, but both are presented to enable a better understanding of how IPF works.

constraint) and 12 in zone 1. We can measure the size of this difference using measures of *goodnes of fit*. A simple measure is total absolute error (TAE), calculated in this case as `sum(abs(cons[1,] - ind_agg))`: the sum of the positive differences between cell values in the individual and aggregate level data.

The purpose of the *reweighting* procedure in spatial microsimulation is to minimise this difference (as measured in TAE above) by adding high weights to the most representative individuals.

3.6 Reweighting using IPF in R

How representative each individual is of each zone is determined by their *weight* for that zone. If we have `nrow(cons)` zones and `nrow(ind)` individuals (3 and 5, respectively, in SimpleWorld) we will create 15 weights. Let us create an empty weight matrix, ready to be filled with numbers calculated through the IPF procedure:

```
weights <- matrix(data = NA, nrow = nrow(ind), ncol = nrow(cons))
```

3.7 Reweighting with ipfp

It is possible to perform IPF much faster and with less code than illustrated above using the **ipfp** R package. The `ipfp` command that implements the IPF algorithm in the C language, as illustrated below on the same dataset:

```
library(ipfp) # load the ipfp library after: install.packages("ipfp")
cons <- apply(cons, 2, as.numeric) # convert matrix to numeric data type
ipfp(cons[1,], t(ind_cat), x0 = rep(1,nrow(ind))) # run IPF
```

```
## [1] 1.228 1.228 3.544 1.544 4.456
```

It is impressive that the entire IPF process which took dozens of lines of code in pure R has been condensed into two lines of code: one to convert the input constraint dataset to `numeric`[6] and one to perform the IPF operation itself. Note also that although we did not specify how many iterations to run, the above command ran the default of `maxit = 1000` iterations, despite convergence happening after 10 iterations. This can be seen by specifying the `maxit` and `verbose` arguments in ipfp, as illustrated below (only the first line of R output is shown):

```
ipfp(cons[1,], t(ind_cat), rep(1, nrow(ind)), maxit = 20, verbose = T)
```

```
## iteration 0:  0.141421
## iteration 1:  0.00367328
```

Notice also that with `ipfp` the categorical `0,1` version of the individual-level data is transposed to represent the individual-level data instead of the `ind_agg` object used previously.

To extend this process to all 6 zones we can wrap the line beginning `ipfp(...)` inside a `for` loop, saving the results each time into the weight variable we created earlier:

```
for(i in 1:ncol(weights)){
  weights[,i] <- ipfp(cons[i,], t(ind_cat), x0 = rep(1, nrow(ind)))
}
```

[6] The integer data type fails because C requires `numeric` data to be converted into its *floating point* data class.

To make this process even faster and more concise (although potentially less clear), however, it is best use R's internal `for` loop via the `apply` function:

```
weights <- apply(cons, MARGIN = 1, FUN =  function(x) ipfp(x, t(ind_cat), x0 = rep(1,nrow(ind))))
```

Note that this function generates the same result as the `for` loop solution. What is happening here is that we are iterating through each row (hence `MARGIN = 1`: `MARGIN = 2` would signify column-wise iteration) of the `cons` object. For each row of data (internally represented by `x`) we applying the `ipfp` function.

3.8 Combinatorial optimisation

3.9 Integerisation

4 CakeMap: spatial microsimulation in the wild

By now we have developed a good understanding of what spatial microsimulation is, its applications and how it works, in terms of the underlying theory and its implementation in R. However, we have yet to see how the method can be applied *in the wild*, on *real* datasets.

"This spatial microsimulation technique seems useful, but how can I use these methods on *my* data?" This is a question many readers will be asking at this stage. The purpose of this chapter is to answer the question, as specifically as possible, to enable the information and code provided in this book to be translated directly into new research in the reader's own field.

The chapter is based on a hypothetical use of spatial microsimulation: to estimate cake consumption in different parts of a city...

4.1 Preparing the input data

Often spatial microsimulation methodology is presented in a way that suggests the data arrived in a near perfect state, ready to be inserted directly into a spatial microsimulation model. This is rarely the case. Usually, one must spend time translating the data into a suitable format, re-coding categorical variables and column names, binning continuous variables and subsetting from the microdataset. All of this can easily take as long as the analysis stage, so it is important to think carefully about strategies for data cleaning before undertaking a complex project (Wickham, 2014). Fortunately R is an accomplished tool for data cleaning (Kabacoff, 2011). To learn about the data cleaning steps that may be useful to your data, we start from the beginning in this section, with a real (anonymised) dataset that was downloaded from the internet.

4.2 Performing IPF on CakeMap data

4.3 Integerisation

4.4 Validation

4.5 Visualisations

4.6 Analysis and interpretation

5 Appendix: Getting up-to-speed with R

As mentioned in Chapter 1, R is a general purpose programming language focussed on data analysis and modelling. This small tutorial aims to teach the basics of R, from the perspective of spatial microsimulation research. It should also be useful to people with existing R skills, to re-affirm their knowledge base and see how it is applicable to spatial microsimulation.

R's design is built on the idea that "everything is an object and everything that happens is a function". It is a *vectorised*, *object orientated* and *functional* programming language (Wickham, 2015). This means that R understands vector algebra, all data accessible to R resides in a number of named objects and that a function must be used to modify any object. We will look at each of these in some code below.

5.1 R understands vector algebra

A vector is simply an ordered list of numbers (Beezer, 2008). Imagine two vectors, each consisting of 3 elements:

$$a = (1, 2, 3); b = (9, 8, 6)$$

To say that R understands vector algebra is to say that it knows how to handle vectors in the same way a mathematician does:

$$a + b = (a_1 + b_1, a_2 + b_2, c_3 + c_3) = (10, 10, 9)$$

This may not seem remarkable, but it is. Most programming languages are not vectorised, so they would see $a + b$ differently. In Python, for example, this is the answer we get:[7]

```
a = [1,2,3]
b = [9,8,6]
print(a + b)
```

```
## [1, 2, 3, 9, 8, 6]
```

In R, the operation *just works*, intuitively:

```
a <- c(1, 2, 3)
b <- c(9, 8, 6)
a + b
```

```
## [1] 10 10  9
```

This conciseness is clearly very useful in spatial microsimulation, as numeric variables of the same length are common (e.g. the ages of all simulated individuals in a zone) and can be acted on with a minimum of effort from the researcher.

5.2 R is object orientated

In R, everything that exists is an object with a name and a class. This is useful, because R's functions know automatically how to behave differently on different objects depending on their class.

To illustrate the point, let's create two objects, each with a different class and see how the function `summarise` behaves differently, depending on the type. This behavior is *polymorphism* (Matloff, 2011):

```
# Create a character and a vector object
char_obj <- c("red", "blue", "red", "green")
num_obj <- c(1, 4, 2, 532.1)

# Summary of each object
summary(char_obj)
```

```
##    Length     Class      Mode
##         4 character character
```

```
summary(num_obj)
```

[7] We can get the right answer in Python, by typing the following: `import numpy; a=numpy.array([1,2,3]); b=numpy.array([9,8,6]); a+b`.

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     1.0     1.8     3.0   135.0   136.0   532.0
```

```
# Summary of a factor object
fac_obj <- factor(char_obj)
summary(fac_obj)
```

```
##  blue green   red
##     1     1     2
```

In the example above, the output from `summary` for the numeric object `num_obj` was very different from that of the character vector `char_obj`. Note that although the same information was contained in `fac_obj` (a factor), the output from `summary` changes again.

Note that objects can be called almost anything in R with the exceptions of names beginning with a number or containing operator symbols such as `-`, `^` and brackets. It is good practice to think about what the purpose of an object is before naming it: using clear and concise names can save you a huge amount of time in the long run.

5.3 Subsetting in R

R has powerful, concise and (over time) intuitive methods for taking subsets of data. Using the SimpleWorld example we loaded in C4, let's explore the `ind` object in more detail, to see how we can select the parts of an object we are most interested in. As before, we need to load the data:

```
ind <- read.csv("data/SimpleWorld/ind.csv")
```

Now, it is easy from within R to call a single individual (e.g. individual 3) using the square bracket notation:

```
ind[3,]
```

```
##   id age sex
## 3  3  35   m
```

The above example takes a subset of `ind` all elements present on the 3rd row: for a 2 dimensional table, anything to the left of the comma refers to rows and anything to the right refers to columns. Note that `ind[2:3,]` and `ind[c(3,5),]` also take subsets of the `ind` object: the square brackets can take *vector* inputs as well as single numbers.

We can also subset by columns: the second dimension. Confusingly, this can be done in four ways, because `ind` is an R `data.frame`[8] and a data frame can behave simultaneously as a list, a matrix and a data frame (only the results of the first are shown):

```
ind$age # data.frame column name notation I
```

```
## [1] 59 54 35 73 49
```

[8]This can be ascertained by typing `class(ind)`. It is useful to know the class of different R objects, so make good use of the `class()` function.

```
# ind[, 2] # matrix notation
# ind["age"] # column name notation II
# ind[[2]] # list notation
# ind[2] # numeric data frame notation
```

It is also possible to subset cells by both rows and columns simultaneously. Let us select query the gender of the 4th individual, as an example (pay attention to the relative location of the comma inside the square brackets):

```
ind[4, 3]
```

```
## [1] f
## Levels: f m
```

A commonly used trick in R that helps with the analysis of individual-level data is to subset a data frame based on one or more of its variables. Let's subset first all females in our dataset and then all females over 50:

```
ind[ind$sex == "f", ]
```

```
##   id age sex
## 4  4  73   f
## 5  5  49   f
```

```
ind[ind$sex == "f" & ind$age > 50, ]
```

```
##   id age sex
## 4  4  73   f
```

In the above code, R uses relational operators of equality (`==`) and inequality (`>`) which can be used in combination using the `&` symbol. This works because, as well as integer numbers, one can also place *boolean* variables into square brackets: `ind$sex == "f"` returns a binary vector consisting solely of `TRUE` and `FALSE` values.[9]

5.3.1 Further R resources

The above tutorial should provide a sufficient grounding in R for begginers to understand the practical examples in the book. However, R is a deep language and there is much else to learn that will be of benefit to your modelling skills. The following resources are highly recommended:

- *An Introduction to R* (Venables et al., 2014) is the foundational introductory R manual, written by the software's core developers. It is terse and covers some advanced topics, but provides an unbeatable introduction to R's behaviour as a language.
- *Advanced R* (Wickham, 2015) is a book that delves into the heart of the R language. It contains many advanced topics, but the introductory chapters are straightforward. Browsing some of the pages on Advanced R's website and trying to answer the questions that open each chapter is an excellent way of testing and improving one's understanding of R.
- *Introduction to visualising spatial data in R* (Lovelace and Cheshire, 2014) provides an introductory tutorial on handling spatial data in R, including the administrative zone data which often form the building blocks of spatial microsimulation models in R.

There are alternatives to R and in the next section we will consider a few of these.

[9] Thus, yet another way to invoke the 2nd column of `ind` is the following: `ind[c(F, T, F)]`! Here, `T` and `F` are shorthand for "TRUE" and "FALSE" respectively.