Spatial microsimulation with R

# 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, software and data size, quality and accessibility mean spatial microsimulation is more accessible than ever. It is now possible to simulate the populations of small administrative areas at the individual-level almost anywhere in the world. This opens new possibilities for a range of applications, not least policy evaluation.

Still, the meaning of spatial microsimulation is still largely unknown. This is partly because the technique is inherently difficult to understand and partly due to researchers themselves. Some of the literature that uses the term is ambiguous, inconsistent about the methods involved. The situation is worsened by work that treats spatial microsimulation as a magical black box or that does not define terms. This ambiguity is partly because 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 field cannot easily be replicated. In today’s age of fast Internet connections, open access datasets and free software, there is little excuse for this. This issue is not unique to the method and is in fact widespread in academia to leading to calls such as that by Sergio Rey for an 'Open Regional Science' ([Rey, 2014](http://link.springer.com/10.1007/s00168-014-0611-7)).

This book encourages reproducibility by providing the tools for its readers to actually *do* spatial microsimulation on realistic data. All the findings presented in this book can be reproduced using code and data in the book's Github repository: [github.com/Robinlovelace/spatial-microsim-book](https://github.com/Robinlovelace/spatial-microsim-book).

Why spend time and effort on reproducibility? The first answer is that reproducibility actually saves time in the long-run, by ensuring more readable code and allowing your results to be easily re-run at a later data. The second reason is more profound. Reproducibility is a prerequisite of falsifiability and falsifiability is the backbone of science (Popper, 1959). The results on non-reproducible research cannot be verified, reducing scientific credibility. These observations inform the book’s practical nature.

This book presents spatial microsimulation is a living, evolving set of techniques rather than a prescriptive formula for arriving at the 'right' answer. This approach that spatial microsimulation largely defined by its user-community, made up of growing number of people worldwide. In terms of reproducibility, this book aims to contribute to the community by encouraging collaboration, innovation and rigour. The book also aims to open the possibilities of spatial microsimulation to more people, with a practical approach that encourages playing with the data and code. As Kabakoff (2011 p. xxii) put it regarding R, "the best way to learn is to experiment".

## Why spatial microsimulation with R?

Software decisions have a major impact on flexibility, efficiency, reproducibility and ease of expressing oneself. Nearly 3 decades ago [Hölm (1987, p. 153)](http://www.jstor.org/stable/10.2307/490448) 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 papers on the subject.

There are many factors that should influence software selection including cost, maturity, features, flexibility and speed. Perhaps most important for busy researchers is ease and speed of writing, adapting and communicating code. R excels in each of these areas.

R is a low-level language compared with programs such as Microsoft Excel and SPSS, which have been used for spatial microsimulation in the past. R offers the researcher great flexibility in designing workflows, analysis stages and even writing one's own functions.

On the other hand, R is *high level* compared with general purpose languages such as C and Python. Instead of writing code to perform statistical operations 'from scratch', R users generally use pre-made functions. To calculate the mean value of variable x, for example, one would need to type 20 characters in Python: float(sum(x))/len(x).[[1]](#footnote-26) In pure R 7 characters are sufficient: mean(x).

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 'out of the box'. In agent-based modelling the statistical analysis of results often occupies more researcher time than running the model itself ([Theile and Grimm, 2012](http://www.sciencedirect.com/science/article/pii/S1364815210000514)); [Theile, 2014](http://www.jstatsoft.org/v58/i02/paper)) and the same applies to spatial microsimulation, making R an ideal choice.

Finally, 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 and improvements are being added all the time. The **ipfp** package, for example, was published in the summer of 2014 and can greatly reduce the computational time taken for a key element of spatial microsimulation process, as we shall see in [*Reweighting with ipfp*](#ipfp).

## Learning the R language

Having learned a little about *why* R is a good tool for the job, it is worth considering at this stage *how* R should be used. It is useful to think of R not as a series of isolated commands, but as an interconnected *language*. As with learning like Spanish or Chinese frequent practice, persistence and experimentation will ensure deep learning.

The most useful practical advice I can give is to organise your workflow. Each project should have its own self-contained folder containing all that is needed to replicate the analysis. This could include the raw (unchanged) input data, a folder containing R code for analysis, a folder for graphical outputs and a folder for data output. To avoid clutter, it is sensible to arrange this content into folders (thanks to Colin Gillespie for this tip):

|-- book.Rmd  
|-- data  
|-- figures  
|-- output  
|-- R  
| |-- load.R  
| `-- parallel-ipfp.R  
`-- spatial-microsim-book.Rproj

The example directory structure above is taken from an early version of this book. It contains the document for the write-up (book.Rmd --- this could equally be a .docx or .tex file) and RStudio's .Rproj file in the *source directory*. The rest of the entities are folders: one for the input data, one for figures generated, one for data outputs and one for R scripts. The R scripts should have meaningful names and contain only code that works and is commented (an additional backup directory could be used to store experimental code). There is no need to be prescriptive in following this structure. However, projects using spatial microdata tend to be complex, so imposing order over your workflow early will likely yield dividends in the long run: you reap what you sow!

The same applies to learning the R language. Fluency allows complex numerical ideas to be described with a small number of keystrokes. If you are a new R user it is therefore worth spending some time learning the R language. To this end [Appendix A](#apR) provides a primer on R from the perspective of spatial microsimulation.

## Typographic conventions

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

* In-line code is provided 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 distinguish 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, a single style, advocated in [Hadley Wickham](http://r-pkgs.had.co.nz/style.html)'s *Advanced R* book ([Wickham, 2014](http://www.crcpress.com/product/isbn/9781466586963)), is followed throughout. Consistent style and plentiful comments will make your code readable by yourself and others for decades to come.

## 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 precursor to a book CRC Press's [R Series](http://www.crcpress.com/browse/series/crctherser) which will be published in summer 2015. Therefore any comments on the code, explanation or contents will be gratefully received.[[2]](#footnote-35)

The structure is as follows:

* A 'no nonsense' and reproducible explanation of spatial microsimulation with [SimpleWorld](#SimpleWorld).
* The basics of spatial microsimulation in R, introducing the main functions and techniques that are used to generate spatial microdata
* [CakeMap](#CakeMap), a larger and more involved example using real data
* [Next steps](#NextSteps), an introduction to the subsequent steps that can be taken after the spatial microdataset has been generated.

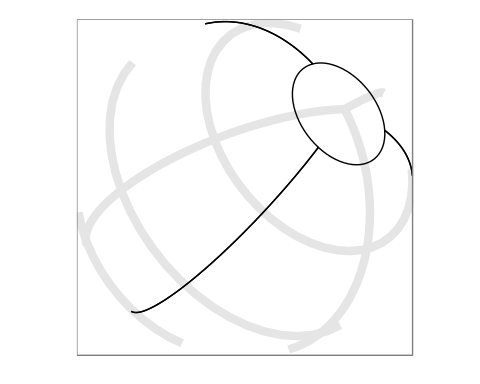
The majority of this material is new and will contribute to a textbook on spatial microsimulation. Please contact [r.lovelace@leeds.ac.uk](mailto:r.lovelace@leeds.ac.uk) with any feedback about the book. Alternatively, if you have a GitHub account, feel free to report any issues online and contribute directly to the project by editing this documents source code: [github.com/Robinlovelace/spatial-microsim-book/blob/master/book-cambridge.Rmd](https://github.com/Robinlovelace/spatial-microsim-book/blob/master/book-cambridge.Rmd).

# An illustrated example from SimpleWorld

To see the link between the methodology introduced [*later in the book*](#Smsim1) and the various real-world applications, let's take a look at a simple example of the kind of situation where spatial microsimulation is useful, to help bridge the gap between method and application.

We'll use an imaginary world called SimpleWorld, consisting of only 3 zones that cover the entirety of the SimpleWorld sphere ([Figure 1](fsimple1)).

##   
## Attaching package: 'ggplot2'  
##   
## The following object is masked \_by\_ '.GlobalEnv':  
##   
## mpg



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.

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

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

Next, imagine a more detailed dataset about 5 of SmallWorld's 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. This is typical of individual-level survey data. Spatial microsimulation tackles this issue 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 weights[5,3], the 5th row and 3rd column in the matrix weights. 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 [a subsequent section](#Smsim1)'.

A more useful output from spatial microsimulation is what we refer to as *spatial microdata*. 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*?

Even though the datasets are tiny in SimpleWorld, we have already generated some useful output. We can estimate, for example, the average income in each zone. Furthermore, we could create an estimate of the *distribution* of income in each area. Although these estimates are unlikely to be very accurate due to the paucity of data, the methods could be very useful if performed on larger datasets from the RealWorld (planet Earth). Finally, the spatial microdata presented in the above table can be used as an input into an agent-based model (ABM). Assuming the inhabitants of SimpleWorld are more predictable than those of RealWorld, the outputs from such a model could be very useful indeed, for example for predicting future outcomes of current patterns of behaviour.

In addition to clarifying the advantages of spatial microsimulation, the above example also flags some limitations of the methodology: 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](#svalidation).

# Preparing input data

The aim of this chapter is to guide you through all the steps of spatial microsimulation in 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.

## Preparing input data

This chapter focuses on the input datasets needed for spatial microsimulation. Correctly loading, manipulating and assessing these datasets will be critical to the performance of your models and the ease of modifying them to include new inputs. This chapter also provides the basis for chapter we perform spatial microsimulation.

As with most spatial microsimulation models, this example consists of a non-geographical individual-level dataset and a series of geographical zones.

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 original 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.

## Loading input data

Real-world individual-level data may be provided in a variety of formats but ultimately needs to be loaded into R as a *data frame* object.

In this case the dataset is loaded from a .csv file:

# Load the individual-level data  
ind <- read.csv("data/SimpleWorld/ind.csv")   
class(ind) # verify the data type of the object

## [1] "data.frame"

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

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

The results of the previous operations are encouraging. The total population is the same for each constraint overall and for each area (row) for both constraints. If the total populations between constraint variables do not match (e.g. because the sample population is different) this is problematic. Appropriate steps to normalise the errant constraint variables are described in [the CakeMap chapter](#CakeMap).

## Subsetting to remove excess information

In the above code, data.frame objects containing precisely the information required for the next stage were loaded. More often, superfluous information will need to be removed from the data and subsets taken. It is worth removing superfluous variables earl, to avoid over-complicating and slowing-down the analysis. If ind had 100 variables of which only the 1st, 3rd and 4th were of interest, for example, the following command could be used to update the object, retaining only the relevant variables: ind <- ind[, c(1, 3, 4)]. Alternatively, ind$age <- NULL removes the age variable.

Although ind is small and simple it will behave in the same way as a much larger dataset, providing opportunities for testing subsetting syntax in R. It is common, for example, to take a subset of the working *population base*: those aged 16 and 74 in full-time employment. Methods for doing this are provided in the [the Appendix on subsetting](#subsetting).

## 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():[[3]](#footnote-46)

# 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.

names(cons)

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

## 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

## '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.[[4]](#footnote-50)

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 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.

# Spatial microsimulation in R

In this chapter we progress from loading and preparing the input data to running a spatial microsimulation model. The SimpleWorld data, loaded in the previous chapter, is used. Being small and simple, the example enables understanding the process on a 'human scale' and allows experimentation without the worry of overloading your computer. However, the methods apply equally to larger and more complex projects. Therefore practicing the basic principles and methods of spatial microsimulation in R is the focus of this chapter. Time spent mastering these basics will make subsequent steps much easier.

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))  
dim(weights) # the dimension of the weight matrix: 5 rows by 3 columns

## [1] 5 3

## IPF in R

One of the simplest ways to allocate the individual-level data loaded in the previous chapter to the three zones of SimpleWorld is using iterative proportional fitting (IPF). IPF is an established technique with a long history. Interested readers are directed towards Lovelace and Ballas ([2012](http://www.sciencedirect.com/science/article/pii/S0198971513000240)) and Pritchard and Miller ([2012](http://link.springer.com/article/10.1007%2Fs11116-011-9367-4)) for recent work in this area, which contain links to the theory and method underlying the method.

## 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[[5]](#footnote-56) 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 a *transposed* (via the t() function) version of the individual-level data (ind\_cat) is used in ipfp to represent the individual-level data, instead of the ind\_agg object used in the pure R version. To prevent having to transpose ind\_cat every time ipfp is called, save the transposed version:

ind\_catt <- t(ind\_cat) # save transposed version of ind\_cat

Another object that can be saved prior to running ipfp on all zones (the rows of cons) is rep(1, nrow(ind)), simply a series of ones. We will call this object x0 as it's argument name representing the starting point of the weight estimates in ipfp:

x0 <- rep(1, nrow(ind)) # save the initial vector

To extend this process to all three 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,], ind\_catt, x0, maxit = 20)  
}

To make this process even more concise (albeit less clear to R beginners), we can use R's internal for loop: apply:

weights <- apply(cons, 1, function(x) ipfp(x, ind\_catt, x0, 20))

In the above code R iterates through each row (hence the second argument MARGIN being 1: MARGIN = 2 would signify column-wise iteration). Thus ipfp is applied to each zone in turn. The speed savings of writing the function with different configurations are benchmarked in 'parallel-ipfp.R' in the 'R' folder of the book project directory. This shows that reducing the maximum iterations of ipfp from the default 1000 to 20 has the greatest performance benefit.[[6]](#footnote-57) To make the code run faster on large datasets, a parallel version of apply called parApply can be used. This is also tested in 'parallel-ipfp.R'.

## Combinatorial optimisation

## Integerisation

# 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?" The purpose of this chapter is to answer this question using a real dataset. This will to enable the information and code provided in this book to be translated directly into the reader's field of interest.

The chapter is based on a hypothetical use of spatial microsimulation to arrive at an important result: estimated cake consumption in different parts of Leeds, UK. The example is deliberately rather absurd, hopefully making the technique more memorable. The steps are presented in a generalisable way, easing the transfer of the method to new datasets.

## 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](http://vita.had.co.nz/papers/tidy-data.html)). Fortunately R is an accomplished tool for data cleaning ([Kabacoff, 2011](http://www.manning.com/kabacoff/)). 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.

## Performing IPF on CakeMap data

In the CakeMap example we use only the ipfp reweighting strategy as it is more concise, generalisable and faster than [IPF in pure R](#IpfinR). On a modern laptop, the ipfp method was found to be *almost 40 times faster* than the 'IPFinR' method. These tests can be run using the microbenchmark() commands found towards the end of 'CakeMap.R'.[[7]](#footnote-65)

## Integerisation

## Validation

## Visualisations

## Analysis and interpretation

# Appendix: Getting up-to-speed with R

As mentioned in [Chapter 1](#Introduction), 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, 2014](http://www.crcpress.com/product/isbn/9781466586963)). 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.

## R understands vector algebra

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

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

This may not seem remarkable, but it is. Most programming languages are not vectorised, so they would see differently. In Python, for example, this is the answer we get:[[8]](#footnote-73)

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.

## 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)

## 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.

## Subsetting in R

R has powerful, concise and (over time) intuitive methods for taking subsets of data. Using the SimpleWorld example we loaded in [*Data preparation*](#DataPrep), 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[[9]](#footnote-76) 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

# 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.[[10]](#footnote-77)

### Further R resources

The above tutorial should provide a sufficient grounding in R for beginners 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, 2014](http://www.crcpress.com/product/isbn/9781466586963)) 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](http://adv-r.had.co.nz/) 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.

1. 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). [↑](#footnote-ref-26)
2. Feedback can be left via email to [r.lovelace@leeds.ac.uk](mailto:r.lovelace@leeds.ac.uk) or via the project's GitHub page. [↑](#footnote-ref-35)
3. 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](http://en.wikipedia.org/wiki/ISO_31-11) for more information. [↑](#footnote-ref-46)
4. As we shall see in [a subsequent section](#ipfp), 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. [↑](#footnote-ref-50)
5. The integer data type fails because C requires numeric data to be converted into its *floating point* data class. [↑](#footnote-ref-56)
6. These tests also show that any speed gains from using apply instead of for are negligible, so whether to use for or apply can be decided by personal preference. [↑](#footnote-ref-57)
7. The second of these benchmarks depends on the old smsim-course, the repository of which needs to be downloaded (from [github.com/Robinlovelace/smsim-course](https://github.com/Robinlovelace/smsim-course)) and saved to the user's computer. [↑](#footnote-ref-65)
8. 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. [↑](#footnote-ref-73)
9. 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. [↑](#footnote-ref-76)
10. 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. [↑](#footnote-ref-77)