Introducing spatial microsimulation with R: a practical

Lovelace, Robin
r.lovelace@leeds.ac.uk

December 25, 2014

Contents

1 Foundations

This practical teaches the basic theory and practice of 'spatial microsimulation' using the popular free software package R. The term microsimulation means different things in different disciplines, so it is important to be clear at the outset what we will and will not be covering.

We will be learning how to create *spatial microdata*, the basis of all spatial microsimulation models, using *iterative proportional fitting* (IPF). IPF is an efficient method for allocating individuals from a non-spatial dataset to geographical zones, analogous to the 'Furness method' in transport modelling, but with more constraints. There are other ways of generating spatial microdata but, as far as the author is aware,[1] this is the most effective and flexible for many applications. An alternative approach using the open source 'Flexible Modelling Framework' program is described in detail, with worked examples, by **?**.

We will not be learning 'dynamic spatial microsimulation' (Ballas et al., 2005): once the spatial microdata have been generated and *integerised*, it is up to the user how they are used — be it in an agent based model or as a basis for estimates of income distributions at the local level or whatever.

We thus define spatial microsimulation narrowly in this tutorial as the process of generating *spatial microdata* (more on this below). The term can also be used to describe a wider approach that harnesses individual-level data allocated to zones for investigating phenomena that vary over space and between individuals such as income inequality or energy overconsumption. In both cases, the generation of spatial microdata is the critical element of the modelling process so the skills learned in this tutorial will provide a firm foundation for further work.

One of the tricky things about spatial microsimulation for newcomers is its use of specialist language. It is important to know exactly what is meant by 'spatial microdata' and other technical terms. To this end we have created a glossary that provide succinct definitions (see Section 5). Any term that is *italicised* in the text has a glossary entry.

1.1 Prerequisites for this practical

This practical uses the statical analysis and modelling software R. We suggest you install and take a look at this powerful program before getting started. We recommend using R within RStudio Fig. 1, which makes using R much easier. Instructions to install both R and RStudio can be found at rstudio.com.

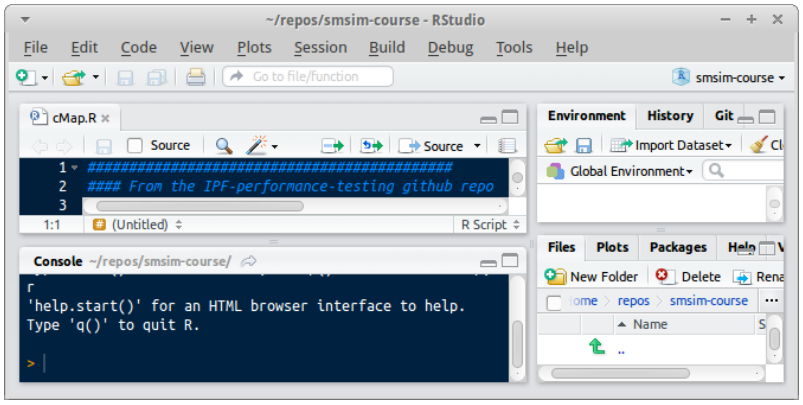

Figure 1: The RStudio user interface. Note the project title 'smsim-course' in the top right and the 'Files' tab at the top of the left hand window.

The other prerequisite for the course is downloading the example data. These can be downloaded in a single zip file which can be

[1] There is much need to compare different methods of generating spatial microdata, for example by building on the work of (Harland et al., 2012). A project for doing these tests in a transparent way has been started at github.com/Robinlovelace/IPF-performance-testing and there are probably many opportunities for improving the computational efficiency of the code presented in this course — please get in touch if you would like to contribute.

found on the course's GitHub repository:[2]
github.com/Robinlovelace/smsim-course. Click on the "Download ZIP" button to the right of this page and extract the folder into your desktop or other suitable place on your computer.

Once the folder has been successfully extracted open it in your browser and take a look around. You will find a number of files and two sub-folders entitled 'data', and 'figures'. When you get to the sections that use R code, it is useful for R to operate from within the smsim-course-master folder. Probably the best way to set this up is to open the file 'smsim-course.Rproj' from within RStudio (Fig. 1). Try this now and click on the 'Files' tab in the bottom right hand window of RStudio. Before using the power of R in RStudio it's worth understanding a bit about 'IPF', the algorithm we will use to generate the synthetic population or 'spatial microdata' (Fig. 2).

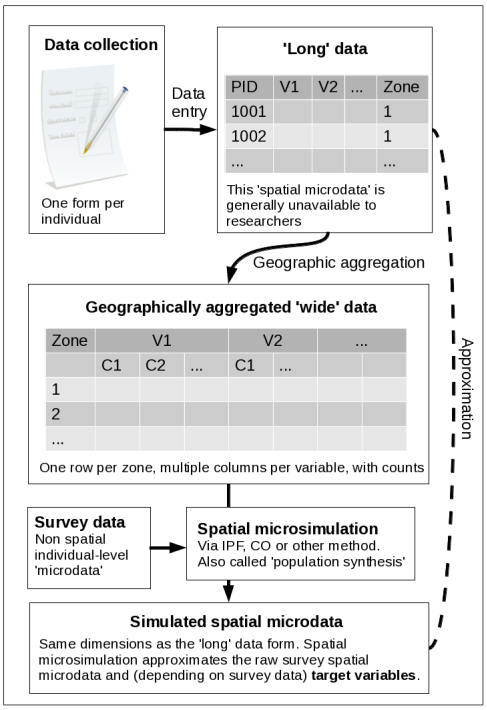

Figure 2: Schema of iterative proportional fitting (IPF) and combinatorial optimisation in the wider context of the availability of different data formats and spatial microsimulation.

1.2 Learning by doing

As Kabacoff (2011, xxii) put it regarding R, "the best way to learn is to experiment" and the same applies to spatial microsimulation. We believe you will learn the technique/art best not by reading about it, but by doing it. Mistakes are inevitable in any challenging task and should not be discouraged. In fact it is by making blunders, identifying and then correcting them that many people learn best. Think of someone learning to skate: no one ever picks up a skateboard for the first time being able to 'surf the sidewalks'. It takes time, patience and plenty of falls before you master the art. The same applies to spatial microsimulation.

Spatial microsimulation works by taking *microdata* at the individual level and using aggregate-level constraints to allocate these individuals to zones. The two main methods are *deterministic reweighting* and *combinatorial optimisation*. This practical takes the former approach using a

[2]GitHub is used to serve this .zip file because if anything changes in the tutorial, the .zip file from the same location will automatically be updated. The other advantages of GitHub are its visibility — a pdf of this tutorial, for example, is kept up-to-date at github.com/Robinlovelace/smsim-course in the 'handout.pdf' file — accessibility and encouragement of evolution: to contribute to this project all one needs to do is 'fork' it.

process called *iterative proportional fitting* (IPF). IPF is used to increase the weights of individuals who are representative of the target area and reduce the weights of individuals who are relatively rare (Lovelace and Ballas, 2013). The output is a *spatial microdataset*.

A schematic of the process is shown in Fig. 2. Take a look at the image and think about the process. But we don't want to get bogged down in theory or applications in this course: we want to 'get our hands dirty'. Next we will do just that, by applying the process to some example data.

1.3 Input data

Let's start with two very basic datasets. To aid understanding, we will primarily do the reweighting by hand to understand the process before automating it on the computer. As outlined in Figure 2, there are two input files required to perform spatial microsimulation:

- Survey data - information about individuals
- Geographically aggregated ('wide') data - typically aggregated Census tables

Table 1 shows 5 individuals, who are defined by two constraint variables: age and sex. This is analogous to the survey data shown in Figure 2. Table 2 presents this same data in aggregated form, whose margin totals can be used in the IPF procedure described shortly. Note that individual 3 is highlighted in blue and bold for future reference.

Table 1: Hypothetical input microdata ('survey data') (the original weights set to one). The bold and blue value is used subsequently for illustrative purposes.

Individual	Age	Sex	Weight
1	59	Male	1
2	54	Male	1
3	35	Male	**1**
4	73	Female	1
5	49	Female	1

Table 2: The aggregated results of the weighted microdata set (m — think "m" for Modified or siMulated). Note, these values are updated by each new set of weights.

Marginal totals		j		
	Age/sex	Male	Female	T
i	Under-50	**1**	1	**2**
	Over-50	2	1	3
	T	3	2	5

Table 3 contains data for a single area. This is analogous to the geographically aggregated 'wide' data shown in Figure 2.

Table 3: Small area constraints (geographically aggregated or 'wide' input data). Note that although individual 3 fits into both "under 50" and "male" categories, only the value of the former is highlighted as age is the first constraint.

Constraint ⇒	i		j	
Category ⇒	i_1	i_2	j_1	j_2
Area ⇓	Under-50	Over-50	Male	Female
1	**8**	4	6	6

Now, it will be recognised by the astute reader that the data in Tables 2 and 3 are incompatible; they are two matrices with different dimensions (3×3 and 1×4 respectively). To solve this problem we can rearrange the small area census constraints data (s, presented in Table 3) to match the dimensions of $m(1)$ (table 2). Table 4 illustrates s in a different form. Note that although we are ignorant of the internal values of this matrix, it is of the same dimension as $m(1)$ and the marginal totals (denoted "T" for total) can be directly compared. This is the basis of the method and provides a way to systematically compare individual level data with geographic aggregates.

Table 4: Small area constraints (matrix s) re-arranged to be the same dimension as m (Table 2). "s" here stands for Small area data or conStraint. Note only the marginal totals are known. The internal cell values (represented by "?") must be estimated by spatial microsimulation.

Marginal totals		j		
	Age/sex	Male	Female	T
i	Under-50	**?**	?	**8**
	Over-50	?	?	4
	T	6	6	12

With our input data in place, we can begin our first iteration.

1.4 The IPF equation

Using the tables m and s presented above we readjust the weights of the individuals so that:

- Individuals who are rare in the geographic zone of interest are given a smaller weight (we need fewer of them).
- Individuals who are common in the zone of interest are given a larger weight (we need more of them).
- Their sum equals the total population of the geography of interest (12 in our example).

For each constraint the current weight (Table 2) is multiplied by the respective total from the geographically aggregated (census) table (Table 3) and divided by the corresponding marginal total of the survey data (see Table 2). For example, using the age constraint the new weight of individual 3 is calculated based on the following numbers: the initial weight is set as 1, the number of individuals under 50 in the geographically aggregated (Census) data is 8 (see Table 3) and the number of individuals under 50 in the survey data is 2 (see Table 2). Thus the new weight is $1 \times \frac{8}{2}$. These figures are highlighted in **bold and blue** in the preceding tables. The same process is used to calculate the new weights for all individuals — remember we are only constraining by age for now. This process of *reweighting* is done one constraint at a time and can be described more succinctly in the form of a word equation Eq. (1), and more generally by Eq. (2) for constraint i (age):

$$Weight_{New} = Weight_{Current} \times \frac{Census_{Constraint\,Total}}{Survey_{Constraint\,Total}} \tag{1}$$

$$w(n+1)_{ij} = \frac{w(n)_{ij} \times sT_i}{mT(n)_i} \tag{2}$$

where $w(n+1)_{ij}$ is the new weight for individuals with characteristics i (age, in this case), and j (sex), $w(n)_{ij}$ is the original weight for individuals with these characteristics, sT_i is element marginal total of the small area constraint, s (Table 3) and $mT(n)_i$ is the marginal total of

category j of the aggregated results of the weighted microdata, m (Table 2). n represents the constraint number.

Do not worry about completely understanding the above procedure for now, its meaning will probably take time to sink in. More important is implementing it.

Follow the blue emboldened values in tables 1 to 4 to fill in the question marks in Table 5 and calculate the new weight of individual 3, a male under 50 years of age. To validate your calculation, the sum of weights (W_1 to W_5) should equal the population of the zone (12, in our example). This is a useful check.

Table 5: Reweighting the hypothetical microdataset in order to fit Table 3.

Individual	Sex	age-group	Weight	New weight, w(2)
1	Male	Over-50	1	$1 \times 4/3 = \frac{4}{3}$
2	Male	Over-50	1	$1 \times 4/3 = \frac{4}{3}$
3	Male	Under-50	1	$1 \times \frac{\mathbf{8}}{\mathbf{?}} = \mathbf{?}$
4	Female	Over-50	1	$1 \times 4/3 = \frac{4}{3}$
5	Female	Under-50	1	$1 \times \frac{8}{2} = 4$
Total			5	12

1.5 Re-aggregation after the first constraint

After creating $Weight_{New}$ for all possibilities of the first constraint (under 50 and 50+ for age), the individual level data is re-aggregated before the next weight can be calculated using the next constraint.

Re-aggregating the individual-level data — to compare the marginal totals with the constraint tables ready for the second constraint — will result in Table 6. To generate this table, the new weights ($w(2)$, presented in Table 5) are multiplied by the number of individuals in each category. Thus, to calculate the new estimate for the number of males we multiply the weight of males under 50 (4 — follow the emboldened values above) by the number of individuals in this category (1) and add the corresponding values for males over 50 ($\frac{4}{3}$ and 2). This is as follows (see the bold value in the bottom left of Table 6):

$$\sum_{i=male} m(2) \; = \; \mathbf{4} \times \mathbf{1} + \frac{4}{3} \times 2 \; = \; \mathbf{4} + \frac{8}{3} \; = \mathbf{6}\frac{\mathbf{2}}{\mathbf{3}} \tag{3}$$

After performing this process of re-aggregation (Table 6) for all categories in the **age** constraint,n the next stage is to repeat Eq. (2) for the **sex** constraint to generate a third set of weights, by replacing the i in sT_i and $mT(n)_i$ with j and incrementing the value of n:

$$w(3)_{ij} = \frac{w(2)_{ij} \times sT_j}{mT(2)_j} \tag{4}$$

1.6 Test your understanding: the second constraint

To test your understanding of IPF, try to apply Eq. (4) to the information above and that presented in Table 6. This should result in the following vector of new weights, for individuals 1 to 5. Calculate the correct values and pencil them in in place of the question marks.

$$w(3) = (\frac{6 \times \frac{4}{3}}{6\frac{2}{3}}, \quad \frac{6 \times \frac{4}{3}}{?}, \quad \frac{\mathbf{6 \times 4}}{\mathbf{6\frac{2}{3}}}, \quad \frac{? \times ?}{?}, \quad \frac{4 \times 6}{5\frac{1}{3}}) \tag{5}$$

After simplifying these fractions, the results are as follows. One 'sanity' check of your method here is whether the sum of these weights is still equal to the area's total population of twelve. Test this is true:

$$w(3) = (\frac{6}{5}, \frac{6}{5}, \frac{\mathbf{18}}{\mathbf{5}}, \frac{3}{2}, \frac{9}{2}) \tag{6}$$

What do the weights in w(3) actually mean? They indicate how representative each individual is of the target zone after one iteration of IPF, constraining by age and sex. Individual number 5 has the highest weight because there is only one young female in the survey dataset yet seemingly many in the area in question.

Notice also that after each iteration the fit between the marginal totals of m and s improves.[3]

Table 6: The aggregated results of the weighted microdata set after constraining for age ($m(2)$).

Marginal totals		i		
	Age/sex	Male	Female	T
j	Under-50	**4**	4	8
	Over-50	$\frac{8}{3}$	$\frac{4}{3}$	4
	T	$\mathbf{6\frac{2}{3}}$	$5\frac{1}{3}$	12

1.7 IPF in a spreadsheet

R is an unfamiliar programme to many, so before we move on to producing an R script that can calculate weights, we will carry out an intermediary step in a spreadsheet programme.[4] For small examples like the one we are using, a spreadsheet has the two main advantages that: it is already familiar to many so can act as a bridge to calculating IPF using a computer; and it is easier to see and recall more than one table in a spreadsheet compared to R with R Studio.

Open sms-spreadsheet-exercise.xlxs if you are using Microsoft Excel or sms-spreadsheet-exercise.odt (note the different file types) if you are using LibreOffice/OpenOffice Calc and follow the tasks outlined in the file. The steps are exactly the same as the pen and paper example above to allow you to more easily check your answers; to help with the transition of performing the calculations of IPF in a computer programme; and to aid recall of the procedure through repetition.

2 IPF in R

So far we have implemented IPF by hand and in a spreadsheet. This section explains how the IPF *algorithm* described above is implemented in R, using the same input data. Now we will be working within the RStudio environment with the 'smsim-course' project loaded and the 'simple.R' file open in the top left panel (Fig. 3).

[3] This can be checked by comparing the aggregated weighted individuals with the small area constraints. Total absolute error (TAE), defined as the sum of all differences between simulated and observed marginal totals, improves between $m(1)$ to $m(2)$, falling from 14 to 6 in Table 2 and Table 6 above. TAE for $m(3)$ (not shown, but calculated by aggregating $w(3)$) improves even more, to 1.3. This number would eventually converge to 0 through subsequent iterations, a defining feature of IPF.

[4] This example was produced by Phil Mike Jones — see philmikejones.net.

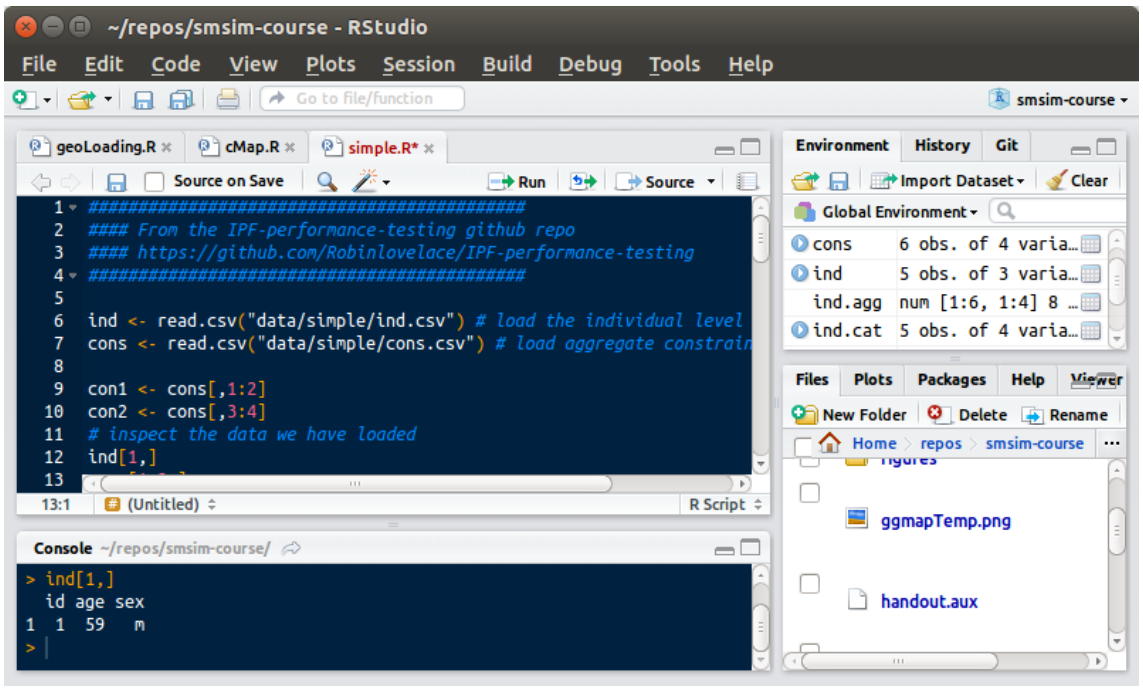

Figure 3: The RStudio user interface with the 'simple.R' file loaded in the top left window and some output displayed in the bottom left window.

Note a couple of things from this figure: there are many explanations of the code and these are commented out by the `#` ('hash') symbol. Also not in the bottom left panel R displays the output of the commands we send it from the script. Hitting the `Ctrl` and `Enter` keys simultaneously from within the upper (script) window will cause R to run these commands, line by line. In the code blocks below, any outputs from R are preceded by '`##`'. Comparing these outputs with the output on your screen will ensure that R is behaving as it should.

We will spend the rest of this section working through the commands contained in this file.[5] The code snippets are taken from the 'simple.R' script file (just a plain text document, like all R scripts) so you can copy and paste. However, it is highly recommended to not look at this file until you need to, towards the end of this section. This is learning by typing, as advocated by Shaw (2013).

2.1 Loading and exploring the input data

The data presented in the above worked example are saved in the 'simple' sub-folder of 'data' as .csv files. To load them in R, we use the following commands:

Listing 1: Loading the input data in R

```
ind <- read.csv("data/simple/ind.csv") # load the individual level data
cons <- read.csv("data/simple/cons.csv") # load aggregate constraints
```

To check that the *data frames* have loaded correctly, simply ask R to print their contents by entering the object's name. To display the individual level data, for example, simply type `ind`:[6]

[5] A longer tutorial is available from Rpubs, a site dedicated to publishing R analyses that are reproducible: rpubs.com/RobinLovelace/5089.

[6] An alternative solution is to click the object's name in 'Environment' in RStudio's top right window.

Listing 2: Checking the contents of the individual level data frame

```
ind  # Show the data frame in R
##  id age sex
##   1  59   m
##   2  54   m
##   3  35   m
##   4  73   f
##   5  49   f
```

Note that the input data is identical to the tables illustrated above for consistency. Instead of constraining for just one zone as we did in the example above, we will fit the data to six zones here. To subset a dataset, add square brackets to the end of the object's name in R. Entering `cons[1:2,]` , for example, should output the first two rows of the constraint variable. Within the square brackets the number before the comma refers to rows (blank means all rows) and the numbers after the comma refer to columns. If the output of this command looks like the text below, congratulations are in order: you have successfully loaded a subset of the constraint dataset.

Listing 3: Printing a subset of the constraint data in R

```
cons[1:2, ]
##    X16.49 X50. m f
## 1       8    4 6 6
## 2       2    8 4 6
```

Note that the top row is identical to Table 3 — we can therefore compare the results of doing IPF on the computer with the results obtained by hand.

Using the subset command just introduced, we must now subset the constraint (`cons`) data frame so we can access each constraint (age or sex) separately. Using your knowledge of the subsetting command, we select columns 1 to 2, and 3 to 4 leaving all rows in place. Give these subsets (constraints) the name `con1` and `con2` respectively:

Listing 4: Subsetting the constraint file

```
con1 <- cons[ ,1:2]
con2 <- cons[ ,3:4]
```

To make the individual level dataset comparable to the aggregate level we must first categorise it and count the number in each category. This job is performed by the R *script* categorise.R, contained in the 'simple' data folder. Run it by entering `source("data/simple/categorise.R")`.[7] This results in the creation of a new object called `ind.cat` in R, which is a matrix of 0s and 1s with same number of columns as the constraint data — try exploring `ind.cat` using the square bracket notation mentioned above.

To check that the script has worked properly, lets count the number of individuals it contains:

The sum of both age and sex variables is 5 (the total number of individuals): it worked! Now the data is in the same form as the constraint variables and we have checked the data makes sense, we can create the 'weight' matrix and begin reweighting the individuals to zones (Listing 6).

[7]Note that the 'categories.R' script must be modified each time it is used with a different dataset.

Listing 5: Output from ind.cat

```
colSums(ind.cat)
##    a16.49    a50.       m       f
##         2       3       3       2
```

Listing 6: Creating the weight and aggregated individual level matrices — see lines 21 and 22 in 'simple.R'

```
weights <- array(1, dim=c(nrow(ind),nrow(cons)))
ind.agg <- matrix(rep(colSums(ind.cat), nrow(cons)), nrow(cons), byrow = T )
```

The code in Listing 6 may not make total sense. Do not worry about this for now, you can always check what is going on later by checking R's documentation (e.g. try entering `?rep`) or by pressing the tab key when entering arguments for R commands in RStudio. For the time being, it is best to press-on with the example and understand the concepts — the details can be looked at later.

2.2 Reweighting by the age constraint

With all of these objects in place, we are ready to begin allocating new weights to the individuals via IPF. Below is written in code the IPF formula presented in Section 1.4. It begins with a couple of *for loops*, one to iterate through each zone (hence the `1:nrow(cons)` argument, which means "from 1 to the number of zones in the constraint data") and one to iterate through each category within constraint 1 (under 50 and over 50 in this case). Before running this code, ensure that all previous commands in the 'simple.R' script have been performed.

Listing 7: Calculating new weights for the individuals based on constraint 1 (age) via IPF — see line 30 and beyond in 'simple.R'

```
for (j in 1:nrow(cons)){
  for(i in 1:ncol(con1)){
 weights[which(ind.cat[,i] == 1),j] <- con1[j,i] / ind.agg[j,i]}}
```

The above code updates the weight matrix by dividing the census constraint (`con1`) by the aggregated version of the individual level data, just as in Section 1.4 above. Further explanation is provided in Section 3.2: for now let's push on to the next block of code, which updates the aggregated data (named `ind.agg` in R) based on the the new weights.

2.3 Re-aggregation

The above line of code multiply the new weights by the flat version of the individual-level data, one zone (i) at a time. This is equivalent to the re-aggregation stage described in Section 1.5 and the result can be checked by printing the first row of our new `ind.agg` object:

To recap, we have updated 'ind.agg' with new values that are constrained by the age variable. This seems to improve the fit between the census and individual-level data as the marginal totals for age are now equal to the aggregate values for age of zone 1. This is further verified in the 'simple.R' script file with a line of code that calculates the Total Absolute Error after each iteration — `sum(abs(ind.agg- cons))` . Check that the error decreases with each iteration (see Section 3.4 for more on model checking and validation).

Listing 8: Re-aggregation of the individual level data based on the new weights (see lines 35 and 36 in 'simple.R')

```
for (i in 1:nrow(cons)){ # convert con1 weights back into aggregates
  ind.agg[i,]   <- colSums(ind.cat * weights[,i])}
```

Listing 9: The new aggregate (marginal) totals for each category after weights have been constrained by age. Compare with Table 6 above.

```
ind.agg[1, ]
##     [1] 8.000000 4.000000 6.666667 5.333333
```

Continue to enter the code contained in 'simple.R' line by line to constrain by the second constraint, sex. Notice that we have done in code what was previously done by hand and in a spreadsheet. Depending on which method you feel most comfortable with, it is worth spending time repeating the previous steps to ensure they make sense.

The other advantage of running the same process 3 times in 3 different ways is that we can cross-compare the results from each. Compare the following listing with Eq. (6), for example: are the weights the same?

Listing 10: W(3) after constraining by age and sex in R. Compare with Eq. (6)

```
weights3[,1] # check the weights allocated for zone 1
##     [1] 1.2 1.2 3.6 1.5 4.5
```

2.4 Iterations

In the above example, we have seen the 'bare bones' of spatial microsimulation using IPF to generate weights from which sample populations can be created. To perform multiple iterations of the same model, we have prepared a slightly more complex script than 'simple.R' called 'simple-iterations.R' that can also be found in the project's root directory. We set the number of iterations manually (just change the value of `num.its`). This happens by calling another script called 'e2.R' which runs the re-weighting process repeatedly, once for each iteration. Notice that only 2 full iterations, the perfect solution is reached (r = 1).

There is great scope for taking the analysis further: some further tests and plots are presented on the on-line versions of this section. The simplest case is contained in Rpubs document rpubs.com/RobinLovelace/6193 and a more complex case (with three constraints) can be found in Rpubs document 5089. For now, however, we progress to a more complex example, CakeMap.

3 CakeMap: A worked example

In this section we will process real data to arrive at an important result: an estimate of the amount of cake eaten in different wards in West Yorkshire. The example is deliberately rather absurd: hopefully this will make the technique more memorable. An attempt has been made to present the method in a generalisable was as possible, allowing users to apply the technique described here to their own data.

The code needed to run the main part of the example is contained within 'cMap.R'. Note that this script makes frequent reference to files contained in the 'cakeMap' folder 'data'.

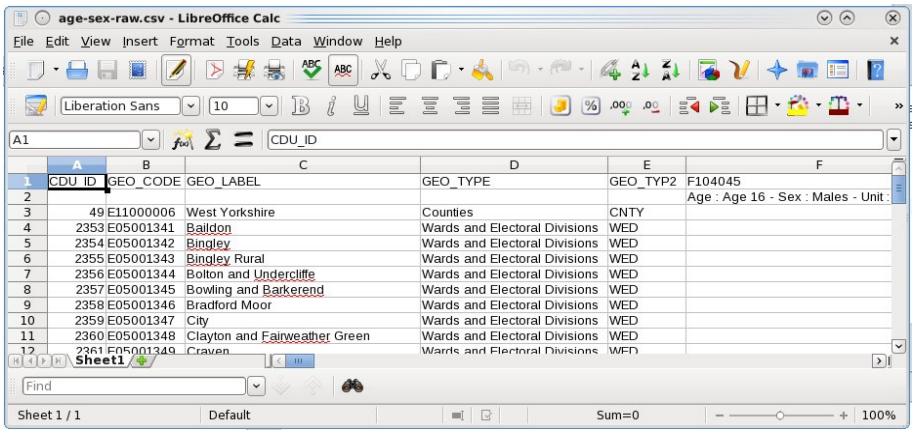

Figure 4: The raw input data for the CakeMap model, downloaded from the InFuse website.

`read.csv("data/cakeMap/ind.csv")`, for example, is used to read the individual level data into R.

Because this example uses real world data (albeit in anonymised form), we will see how important the process of 'data cleaning' is. Rarely are datasets downloaded in exactly the right form for them to be pumped directly into a spatial microsimulation model. For that reason we describe the process of loading and preparing the input data first: similar steps will be needed to use your own data as an input into a spatial microsimulation model.

3.1 Loading and preparing the input data

The raw constraint variables for CakeMap were downloaded from the Infuse website (infuse.mimas.ac.uk/). These, logically enough, are stored in the 'cakeMap/data/' directory as .csv files and contain the word 'raw' so it is clear which files contain the raw data. The file 'age-sex-raw.csv', for example is the raw age and sex data that was downloaded. As the screenshot in Fig. 4 shows, these datasets need to be processed before they can used as an input in our model. 'con1.csv' stands for 'constraint 1' and represents this age/sex data after it has been processed.

To ensure reproducibility in the process of converting the raw data into this 'spatial microsimulation ready' dataset, all of the steps used to clean and rearrange the data have been saved. Take a look at the R script file 'process-age.R' — these are the kinds of steps that the researcher will need to undertake before performing a spatial microsimulation model. Here is not the place to delve into the details of data reformatting; there are resources dedicated to that (Wickham, 2014; Kabacoff, 2011). However, it is worth taking a look at the 'process-age.R' script and the other 'process*' files to see how R can be used to quickly process complex raw data into a form that is ready for use in spatial microsimulation.

The input data generated through this process of data preparation are named 'con1.csv' to 'con3.csv'. For simplicity, all these were merged into a single dataset called 'cons.csv'. All the input data for this section are loaded with the following commands:

Listing 11: Loading the input data for CakeMap (see 'cMap.R')

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

Take a look at these input data using the techniques learned in the previous section. To test your understanding, try to answer the following questions: what are the constraint variables? How many individuals are in the survey microdataset? How many zones will we generate spatial microdata for?

For bonus points that will test your R skills as well as your practical knowledge of spatial microsimulation, try constructing queries in R that will automatically answer these questions.

It is vital to understand the input datasets before trying to model them, so take some time exploring the input. Only when satisfied with your understanding of the datasets (a pen and paper can help here, as well as R!) is it time to move on to generate the spatial microdata using IPF.

3.2 Performing IPF on the CakeMap data

The R script used to perform IPF on the CakeMap data is almost identical to that used to perform the process on the simple example in the previous section. The main difference is that there are more constraints (3 not 2) but otherwise the code is very similar. Confirm this for yourself: it will be instructive to compare the 'simple.R' and 'cMap.R' script files. Note that the former works on a tiny example of 5 input individuals whereas the latter calculates weights for 916 individuals. The code is generalisable, meaning that the IPF process would still work given slightly different inputs. The script would still work if more individuals or zones were added although the code will need to be adjusted to accept differenct constraints. To ensure the process works with different constraints, the file 'categorise.R' must also be.

Note that 'cMap.R' contains 100 lines — longer than the 'simple.R' file. The additional length is due to the third constraint, additional iterations (see line 66 onwards) and some preliminary analysis. The best way to understand what is happening in this script is simply to read through it and run chunks of it in R (remember the `Ctl-Enter` command) to see what happens. Try to 'think like a computer' and imagine what each line of code is doing. This is inevitably difficult at first so it is recommended that users add more comments to lines that cause confusion to help remember what is going on. To help this process, the meaning of some of the trickier and more important lines is explained below. One potentially confusing area is the representation of the `weights` object, used to store the weights after each constraint. To understand what is happening try entering this code into R and seeing what happens:

Listing 12: Creating the initial weight matrix — see lines 28 to 30 in cMap.R

```
# create weights in 3D matrix (individuals, areas, iteration)
weights <- array(dim=c(nrow(ind),nrow(cons),num.cons+1))
weights[,,num.cons+1][] <- 1 # sets initial weights to 1
ini.ws <- weights[,,num.cons+1]
```

In the above code, `weights` is created by the `array` command, which in fact creates many 2 dimensional matrices in a 'data cube'. This is different from the weight object used in the 'simple.R' script, which had only 2 dimensions.

If this is confusing, just remember that the 'square' weight matrix — where each column represents a zone and each row represents an individual — can be returned to by specifying the third dimension. Dimensions in R are specified by the square brackets, so the initial weights can be called by typing `weights[,,4]`. This should result in hundreds of 1s, the initial weights. To print a more manageable-sized output, remember how to subset the other dimensions: `weights[1:3,1:5,4]`, for example, will print the first 3 rows (individuals) and first 5 columns of the 4th matrix. Why do we do it like this? So that we only need one weights object to remember all the previous weights so we can go back and check whether the model fit is improving from one constraint and one iteration to the next.

Now, when the new weights are set after iteration 1, they are saved as follows, within a nested *for loop* that iterates over all zones (j) and constraint categories (i):

Listing 13: Saving the new weights after the first constraint of IPF — see line 43 in 'cMap.R'

```
weights[which(ind.cat[,i] == 1),j,1] <- con1[j,i] /ind.agg[j,i,1]}}
```

Note the heavy use of square brackets in Listing 13. Let's break down each element of the code within these square 'subsetting' brackets:

- `which(ind.cat[,i] == 1)` is a subset of the rows in the weight matrix which correspond to all individuals who belong to category i. (Remember, `ind.cat` is the 'wide' version of the individual level data, containing 1s for all categories to which each individual belongs.)
- `,j,1]` specify the column and constraint respectively. Because we are in a for loop, j simply runs through all categories within the current constraint (age/sex). The 1 is selected because this is the first set of re-weighting (this is analogous to say $w(2)$ — the first new weight — in the IPF equations in the previous section).
- Finally, `con1[j,i] /ind.agg[j,i,1]` is R's implementation of Eq. (2): we divide the desired census values by the current simulated totals for each category based on the individual's current weight.

This explanation should help understand what is going on in other parts of the script file also — for example try printing subsets of the `weights` object after running constraint 1, 2 and then 3 to see how the weights change.

3.3 Integerisation

Integerisation increases the utility of results generated by IPF, such as the weights produced in the cakeMap example. Many of the most useful things we can do with individual-level data can only be done with *whole individuals*. The weight matrices we have generated, however, are *fractional weights*, which could imply, for example that 0.437 of an individual resides in a zone. From the perspective of an agent based (but not aggregated level) model, this is clearly absurd.

Here is not the place to delve into the various methods available for integerisation — see (Lovelace and Ballas, 2013) for detail on the problem. Suffice to say that of the two broad approaches available, deterministic and probabilistic, the latter is found to generate more accurate results.

Two scripts to *integerise* the final weights (referred to as `fw` in the code) of the cakeMap model are provided: 'pp-integerise.R' and 'TRS-integerise.R', which stand for 'proportional probabilities' and 'truncate, replicate, sampe'. These reside in the 'data/cakeMap' folder. We will not describe the theory underlying the method here (see Lovelace and Ballas, 2013 for more). Instead, we will describe a few of the functions contained in these scripts that are the keys to understand how they work. Both scripts begin by creating a series of objects that are used during the integerisation process and to save the results Listing 14.

Listing 14: The new objects that are created in at the beginning of the integerisation scrips (see lines 7 to 8 in 'pp-integerise.R' and 'TRS-integeris.R').

```
intall <- ints <- as.list(1:nrow(cons))
intagg <- cons * 0
```

In the first line of Listing 14, two identical objects are created (note the double use of the `<-` object assignment symbol). These are set objects in the list class, allowing each element to be allocated as any other object. This differs from the stricter 'data.frame' and 'matrix' or 'array' objects we have been using so far. Subsequently in the script, individual elements are referred

to using double square brackets. For example, see `ints[[i]] <-` on line 15 of 'pp-integerise.R'. Using list elements provides flexibility: we can set any list object to be anything we like without worrying about constraints imposed by its class or dimensions.

The key function in the proportional probabilities script is `sample`, used in a for loop over all areas (Listing 15).

Listing 15: The sample function, key to R's implementation of the proportional probabilities integerisation algorithm (see lines 15 and 16 of 'pp-integerise.R').

```
ints[[i]] <- sample(which(fw[,i] > 0), size = sum(con1[i,]),
                    prob=fw[,i], replace = T)
```

The `sample` function in Listing 15 contains four arguments, and the meaning of each is described below:

- The first item (`which(fw[,i] > 0)`) provides the *vector* of numbers or characters from which the sample will be drawn. In this case we are referring to the weights of all individuals for zone i (all weights should be above 0).
- The size argument tells R how many items from the object should be selected. In this case it is simple: the total population of zone i.
- The cleverest thing about R's sampling function is that it allows probabilities of being selected to be assigned to each element. Here we are setting the probability of selection as proportional to the final weight. (Clearly the probability of selection can never be greater than 1, as is the case with many weights; R automatically normalises the values.)
- The final argument tells R that we want to allow repeat sampling: the same individual can be selected more than once. This makes sense in a dataset where wheights can get very large: an individual with a weight of 10.4, for example, should ideally be selected between 10 and 11 times.

Note that 'trs-integerise.R' also uses the sample function, in this case to 'top up' individuals already selected from the 'truncate, replicate' stages of the 'truncate, replicate, sample' method (Lovelace and Ballas, 2013). For more information on implementing TRS in R, refer to this paper and the paper's detailed supplementary information, which provides reproducible results to suggest using TRS for accuracy. (As an aside, the accuracy of each integerisation algorithm was tested on the cakeMap data — showing TRS to yield better results for this application. The results can be reproduced by running the 'integerisation-test.R' script, located in the 'vignettes' folder.)

The most useful result of these integerisation algorithms is the object `intall`. This is list of data frames containing, each containing a population equal to the population of its respective zone. Each has the same variables as are present in the original survey dataset. Make this object easier to analyse, a series of commands is provided (but commented out) at the end of the script to convert the list into a more manageable data.frame object. Try entering `nrow(intall.df)` after running this code (remove the comments first!) We have cloned hundreds of thousands of individuals. Clearly, all this data takes up more RAM on the computer, as can be seen by asking `object.size(intall.df)`. Try comparing this result with the size of the original survey dataset 'ind'. From this point onwards, with the individual-level data in a straightforward dataframe (with the zone number kept in the 'zone' column), the modelling and analysis applications should be comparatively straightforward.

Next we move on to a vital consideration in spatial microsimulation models such as cakeMap: validation.

3.4 Model checking and validation

To make an analogy with food safety standards, openness about mistakes is conducive to high standards (Powell et al., 2011). Transparency in model verification is desirable for similar reasons. The two main strategies are 1) comparing the model results with knowledge of how it *should* perform *a-priori* (model checking) and 2) comparison between the model results and empirical data (validation).

Within the two R scripts we have been using for reweighting the individual-level data so far ('simple.R' and 'cMap.R'), there are *already* commands whose function is to check the results are as expected. Returning to the former, note that on line 24 of 'simple.R', the fit between the survey data and census data is tested with `sum(abs(ind.agg - cons)) # the total absolute error`. This, as the comment implies, calculates the total absolute error (TAE), which is a good measure of the fit between two aggregate-level datasets and which is defined by the following formula:

$$TAE = \sum_{ij} |U_{ij} - T_{ij}| \tag{7}$$

Standardised Total Error is a related measure: $SAE = TAE/P$ where P is the total population of the study area. Thus TAE is sensitive to the number of people within the model. SAE is not — see more on model checking and tests of fin in Lovelace and Ballas (2013)

Note that TAE is calculated twice more in 'simple.R', after each additional constraint has been applied. Work through the code slowly and check how TAE reduces after each constraint and each iteration.

Beyond typos or simple conceptual errors in model code, more fundamental questions should be asked of spatial microsimulation models. The validity of the assumptions on which they are built, and the confidence one should have in the results are important. For this we need external datasets. Validation is therefore a tricky topic — see Edwards and Clarke (2009) for more on this and Section 3.6 for (an albeit unreliable) comparison between estimated cake consumption and external income estimates.

3.5 Visualising the results

Visualisation is an important part of communicating quantitative data, especially so when the datasets are large and complex so not conducive to description with tables or words.

Because we have generated spatial data, it is useful to create a map of the results, to see how it varies from place to place. The code used to do this found in 'cMapPlot.R'. A vital function within this script is 'merge', which is used to add the simulated cake data to the geographic data frame:[8]

Listing 16: The merge function for joining the spatial microsimulation results with geographic data. Compare with Eq. (6)

```
merge(wardsF, wards@data, by = "id")
```

The above line of code by default selects all the data contained in the first object ('wardsF') and adds to it new variables from the second object based on the linking variable (in this case "id"). Also in that script file you will encounter the function `fortify`, the purpose of which is to

[8] 'join' is an alternative to merge from the 'plyr' package also used in the 'cMapPlot.R' script that performs the same task. Assuming 'plyr' is loaded — 'library(plyr)' you can read more about join by entering '?join' in R.

convert the spatial data object into a data frame. More on this process is described in Lovelace and Cheshire (2014). The final map result of 'cakeMapPlot.R' is illustrated Fig. 5.

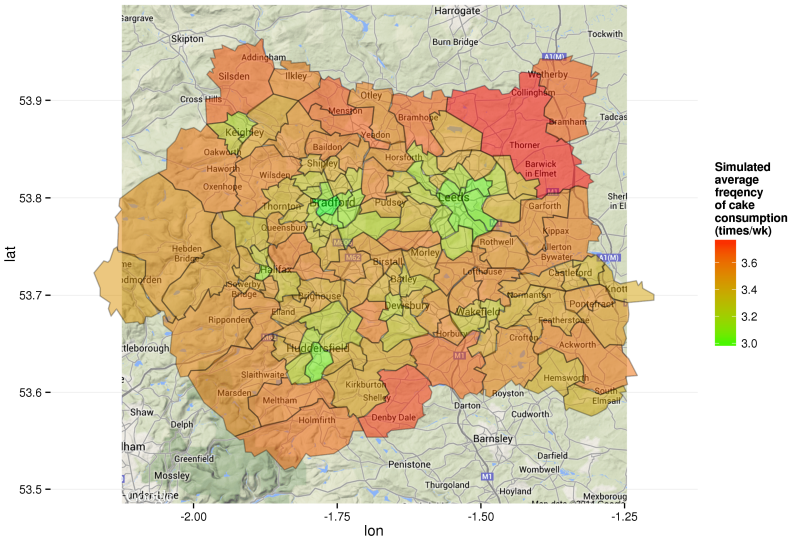

Figure 5: Choropleth map of the spatial distribution of average frequency of cake consumption in West Yorkshire, based on simulated data.

3.6 Analysis

Once a spatial microdataset has been generated that we are happy with, we will probably want to analyse it further. This means exploring it — its main features, variability and links with other datasets. To illustrate this process we will load an additional dataset and compare it with the estimates of cake consumption per person generated in the previous section at the ward level.

The hypothesis we would like to test is that cake consumption is linked to deprivation: More deprived people will eat unheathily and cake is a relatively cheap 'comfort food'. Assuming our simulated data is correct — a questionable assumption but lets roll with it for now — we can explore this at the ward level thanks to a dataset on modelled income from neighbourhood statistics.

Because the income dataset was produced for old ward boundaries (they were slightly modified for the 2011 census), we cannot merge with the spatial dataset based on the new zone codes. Instead we rely on the name of the wards. The code below provides a snapshot of these names and demonstrates how they can be joined using the 'join' function.

Listing 17: The merge function for joining the spatial microsimulation results with geographic data. Compare with Eq. (6)

```
wards@data <- join(wards@data, imd)
summary(imd$NAME %in% wards$NAME)
##        Mode   FALSE    TRUE    NA's
##     logical      55      71       0
```

The above code first joins the two datasets together and then checks the result by seeing how many matches names there are. In practice the fit between old names and new names is quite

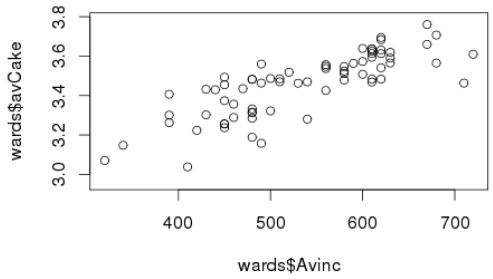

Figure 6: Relationship between modelled average ward income and simulated number of cakes eaten per person per week.

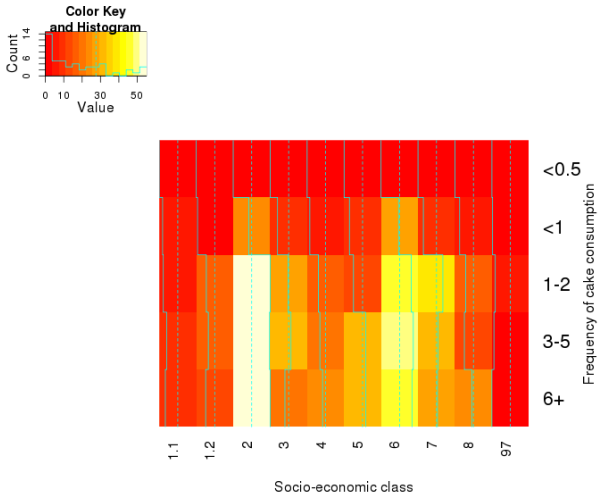

Figure 7: Relationship between social class and cake consumption in the individual level data.

poor: only 71 out of 124. In a proper analysis we would have to solve this problem (e.g. via the command `pmatch` which stands for partial match). For the purposes of this exercise we will simply plot income against simulated cake consumption to gain a feeling what it tells us about the relationship between cake consumption and wealth (Fig. 6).

The question raised by this finding is: why? Not why is cake consumption higher in wealthy areas (this has not been established) but: why has the model resulted in this correlation? To explore this question we need to go back and look at the individual level data. The most relevant constraint variable for income was class. When we look at the relationship between social class and cake consumption in the Dental Health Survey, we find that there is indeed a link: individuals in the highest three classes (1.1, 1.2, 2) have an average cake intake of 3.9 cakes per week whereas the three lowest classes have an average intake of 3.7. This is a relatively modest difference but, when averaging over large areas, it helps explain the result. The class dependence of cake consumption in the Dental Health Survey is illustrated in Fig. 7.

4 Discussion

What have we learned during the course of this practical tutorial? Given that it's geared towards beginners, from one perspective one could say 'not a lot' relative to the many potential applications. The emphasis on understanding of the basics was deliberate: it is only on strong foundations that large and long-lived buildings can be constructed and the same applies to geographical research.

The focus on foundations is also due to the nature of spatial microsimulation research: few people are actively involved in applied work, let alone the development of new software and methods. Specialist skills and understanding are needed to join this research area yet the barriers to entry are high. A central outcome of this course has been to provide an accessible 'way in' to this exciting research area. Specifically, we have learned how to:

- Perform iterative proportional fitting, one of the commonly used techniques for spatial microsimulation, by hand, in a spreadsheet and in R.
- Iterate the process any number of times using pre-made R code.
- Use simple queries and visuals to check the process has worked as expected.
- Process the spatial microdata to create estimates of target variables and join this with output with geographics zones.
- Map the result and begin to look for explanation in the dataset.

An important lesson from the cakeMap example is that spatial microsimulation can be used in careless and misleading ways. The method is powerful, but it has dangers and limitations that must be ackowledged. Many articles on spatial microsimulation omit technical details of the method, let alone code that would allow the findings to be reproduced. Great care must be taken not to mislead.

Returning to cakeMap, we have *not* demonstrated cake consumption to be higher in wealthy areas of Leeds. We have done nothing of the sort! We *have* created spatial microdata for Leeds based on 3 constraint variables. This spatial microdataset is not 'real'. We can have a high level of confidence that the individual attributes constrained by census data are roughly representative of the region. Yet the interrelationship between these constraint variables (e.g. the cross tabulation between age and car ownership) will tend towards the average contained within the survey microdataset and is likely to mask regional variability. In an ward where the proportion of young drivers is very high, for example, our model result is likely to provide underestimates of the numbers of young people owning cars.

The estimates generated for the target variable — cake consumption — are likely to be far from reality. They reflect the relationship between cake consumption and a selection of constraint variables at the national level. Do we have good evidence to suggest that age/sex, socio-economic class and household car ownership are good proxies of cake consumption across different parts of the country? No. So we should treat the results as what they are: an amusing example of what you can do with spatial microsimulation, not a reliable description of the real world. To validate this example we would need to conduct some kind of randomised survey in the target wards to identify whether cake consumption really does vary in the ways described by the model. Most likely we would find that it does not.

This brings us nicely on to the final point of discussion: how you use spatial microsimulation in your own research. This of course is up to you, but during the process of this course we hope that you have picked up some ideas about best practice in the field. These include:

- Ensure, to the extent possible, **reproducibility in your method and findings**. This includes, at a minimum, clear description of the input data, explanation of the method used and the software needed. As illustrated with the cakeMap example, it is now relatively easy to ensure complete reproducibility even in complex analyses. This will not always be possible due to confidentiality of input data. However, the creation of an example dataset and provision of code should always be possible. This is highly recommended as it will greatly help others reproduce your findings (improving the scientific credibility of your research), provide a learning opportunity for yourself and others and increase the probability of other academics citing your work.
- Use spatial microsimulation only when it is the **most appropriate tool for the job**. This

means that if there are alternatives such as geographically weighted regression analysis or analysis of large secondary datasets, these should be considered beforehand. Generating an entirely new individual-level dataset is not to be taken lightly and risks distracting from more grounded research. Thus it should be seen as an *addition to* rather than a *replacement for* more established methods.

- When spatial microsimulation is used, **be transparent about its underlying assumptions and limitations**. Simply forgetting to include the word 'simulated' in the caption of Fig. 5, for example, could lead the reader to believe it is actual cake consumption that is being described. In the case of a cakeMap this may not matter but in areas of public health and the environment the consequences of such oversight could be deadly.

Underlying each of these points is a wider responsibility: to communicate one's research with clarity and transparency. Too much modelling research is shrouded in a cloud of jargon, unstated assumptions and verbose English. If nothing else, this tutorial should provide guidance on how to improve standards in the field and move towards best practice for reproducibility (**?**).

Spatial microsimulation is a powerful tool. Like any powertool, it can acheive very useful results for its user but can also cause great harm if used incorrectly. Think of a pneumatic drill: this could be used to build new public infrastructure such as bicycle paths. It could also, in clumsy hands, be used to destroy existing infrastructure. The same applies to spatial microsimulation: at best it can greatly help out with complex research questions such as the distributional impacts of new transport policies (Lovelace et al., 2014). At worst, it can waste valuable research time, create misleading results and shroud academic research behind an impenetrable wall of jargon.

We have little doubt that the vast majority of people will prefer the former option. This course has hopefully equipped its students with the tools to pursue this lofty aim.

5 Glossary

- **Algorithm**: a series of computer commands which are executed in a well defined order. Algorithms process input data and produce an output.
- **Combinatorial optimisation** is an approach to spatial microsimulation that generates spatial microdata by randomly selecting individuals from a survey dataset and measuring the fit between the simulated output and the constraint variables. If the fit improves after any particular change, the change is kept. Williamson (2007) provides a practical user manual for implementing the technique in code.
- **Data frame**: a type of object (formally referred to as a class) in R, data frames are square tables composed of rows and columns of information. As with many things in R, the best way to understand data frames is to create them and experiment. The following creates a data frame with two variables: name and height:

```
data.frame(name = c("Robin", "Phil"), height.cm = c(172, 174))
```

  Note that each new variable is entered using the command `c()`— this is how R creates objects with the *vector* data class, a one dimensional matrix — and that text data must be entered in quote marks.
- **Deterministic reweighting** is an approach to generating spatial microdata that allocates fractional weights to individuals based on how representative they are of the target area. It differs from combinatorial optimisation approaches in that it requires no random numbers. The most frequently used method of deterministic reweighting is IPF.

- **For loops** are instructions that tell the computer to run a certain set of command repeatedly. `for(i in 1:9) print(i)`, for example will print the value of i 9 times. The best way to further understand for loops is to try them out.
- **Iteration**: one instance of a process that is repeated many times until a predefined end point, often within an *algorithm*.
- **Iterative proportional fitting** (IPF): an iterative process implemented in mathematics and algorithms to find the maximum likelihood of cells that are constrained by multiple sets of marginal totals. To make this abstract definition even more confusing, there are multiple terms which refer to the process, including 'biproportional fitting' and 'matrix raking'. In plain English, IPF in the context of spatial microsimulation can be defined as *a statistical technique for allocating weights to individuals depending on how representative they are of different zones*. IPF is a type of deterministic reweighting, meaning that random numbers are not needed to generate the result and that the output weights are real (not integer) numbers.

6 Acknowledgements

Many thanks to Phil Mike Jones from the University of Sheffield for working through early drafts of this document and providing very useful input to the teaching materials overall. Thank you to Amy O'Neill and Rosie Temple for organising the National Centre for Research Methods (NCRM) course "An Introduction to Spatial Microsimulation using R" and dealing with the high demand. Finally, thanks to the open source software movement overall, for encouraging best practice in reproducible research and making powerful tools such as R accessible to anyone, regardless of income, nationality or status.

7 References

Dimitris Ballas, Danny Dorling, Bethan Thomas, and David Rossiter. *Geography matters: simulating the local impacts of national social policies*. Number 3. Joseph Roundtree Foundation, January 2005. doi: 10.2307/3650139. URL `www.jrf.org.uk/sites/files/jrf/1859352669.pdf`.

Kimberley L Edwards and Graham P Clarke. The design and validation of a spatial microsimulation model of obesogenic environments for children in Leeds, UK: SimObesity. *Social science & medicine*, 69(7):1127–1134, October 2009. ISSN 1873-5347. doi: 10.1016/j.socscimed.2009.07.037. URL `http://www.ncbi.nlm.nih.gov/pubmed/19692160`.

Kirk Harland, Alison Heppenstall, Dianna Smith, and Mark Birkin. Creating Realistic Synthetic Populations at Varying Spatial Scales: A Comparative Critique of Population Synthesis Techniques. *Journal of Artificial Societies and Social Simulation*, 15(1):1, 2012. ISSN 1460-7425. URL `http://jasss.soc.surrey.ac.uk/15/1/1.html`.

Robert Kabacoff. *R in Action*. Manning Publications Co., 2011.

Robin Lovelace and Dimitris Ballas. Truncate, replicate, sample: A method for creating integer weights for spatial microsimulation. *Computers, Environment and Urban Systems*, 41:1–11, September 2013. ISSN 01989715. doi: 10.1016/j.compenvurbsys.2013.03.004. URL `http://dx.doi.org/10.1016/j.compenvurbsys.2013.03.004`.

Robin Lovelace and James Cheshire. Introduction to visualising spatial data in R. *National Centre for Research Methods Working Papers*, 14(03), 2014. URL `https://github.com/Robinlovelace/Creating-maps-in-R`.

Robin Lovelace, Dimitris Ballas, and Matt Watson. A spatial microsimulation approach for the analysis of commuter patterns: from individual to regional levels. *Journal of Transport Geography*, 34(0):282–296, January 2014. ISSN 0966-6923. doi: http://dx.doi.org/10.1016/j.jtrangeo.2013.07.008. URL `http://www.sciencedirect.com/science/article/pii/S0966692313001361`.

Douglas a. Powell, Casey J. Jacob, and Benjamin J. Chapman. Enhancing food safety culture to reduce rates of foodborne illness. *Food Control*, 22(6):817–822, June 2011. ISSN 09567135. doi: 10.1016/j.foodcont.2010.12.009. URL `http://linkinghub.elsevier.com/retrieve/pii/S0956713510004378`.

Zed Shaw. *Learn Python the Hard Way: A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code*. Pearson Education, 2013.

Hadley Wickham. Tidy data. *The Journal of Statistical Software*, 14(5), 2014. ISSN 1548-7660. URL `http://www.jstatsoft.org/v59/i10http://vita.had.co.nz/papers/tidy-data.html`.

Paul Williamson. CO Instruction Manual: Working Paper 2007/1 (v. 07.06.25). Technical Report June, University of Liverpool, 2007.