Supplementary information: a user manual for the integerisation of IPF weights using R

Robin Lovelace
Geography Department
University of Sheffield,
Sheffield,
United Kingdom,
S10 2TN

March 21, 2013

This worked example demonstrates how the methods described in the paper "'Truncate, replicate, sample': a method for creating integer weights for spatial microsimulation" (Lovelace and Ballas, 2013) were conducted in R, a free and open source object-orientated statistical programming language. An introduction to performing iterative proportional fitting (IPF) in R is provided in a separate document.[1] This worked example focuses on methods for converting the results into integer weights, with reference to the code that accompanies this guide. The main aims are to:

1. Introduce R as a user friendly and flexible tool to perform spatial microsimulation and analyse the results;

2. Demonstrate the replicability of the results described in the paper;

3. Encourage unrestricted access to code within the microsimulation community. It is hoped that this will:

[1]This document is titled "Spatial microsimulation in R: a beginner's guide to iterative proportional fitting (IPF)", is available from `http://rpubs.com/RobinLovelace/5089` . A larger project, aimed at optimising R code for IPF applications can be found at `https://github.com/Robinlovelace/IPF-performance-testing` .

- Enhance transparency, repeatability and knowledge transfer within the field;
- Allow others to use, test and further develop existing work, rather than starting from nothing each time, and;
- Allow other researchers to critically assess the four integerisation methods presented in the paper — named simple round, the threshold approach, proportional probabilities and truncate, replicate, sample (TRS) — so they can be improved.

This worked example can be used in different ways, depending on one's aims. The first section shows how the necessary files can be downloaded and loaded into R. Section 2 (Running the Spatial Microsimulation Model) is linked to aim 1, and shows how the microsimulation model, which forms the foundation of the analysis presented in the paper, works. Section 3 (Integerisation in R) is linked to aim 2, and demonstrates how to run each type of integerisation, and display some of the results. Finally, Section 4 (Adapting the Model), illustrates how the code constituting the spatial microsimulation model and integerisation techniques can be adapted to different constraint variables for areas with different population sizes. Aim 3 can be met throughout: we encourage other researchers to experiment with and re-use our code, citing this work where appropriate. To do this, the first stage is to download the dataset and load it into R.

1 Downloading and loading the files into R

Throughout this example, we assume that the data has been downloaded and extracted to a folder titled 'ints-public' onto your desktop and that R is installed on your computer. To download R, visit the project's homepage and follow the instructions. For new R users, it is recommended that a introductory text is acquired and referred to throughout. A range of excellent introductory guides are available online at http://cran.r-project.org/other-docs.html.

To access the files, unzip the file titled 'ints-public.zip', which is available online in the supplementary data.[2] A list of the folders and files contained within the folder 'ints-public' is provided in Table 1.

[2]From here: https://dl.dropbox.com/u/15008199/ints-public.zip

Table 1: Files and folders contained in the worked example folder

Folder	**File**	**Description**
etsim: Spatial microsimulation model based on IPF	Four *.csv files, e.g. age-sex.csv	Constraint variables at MSOA level. Based on scramble census data.
	etsim.R	The spatial microsimulation model resulting in non-integer weights
	cons.R	R script to read constraint variables
	plotsim.R	R script to plot the model output
	USd.cat.r	Script to re-aggregate the results
	Usd.RData	Scrambled survey data based on the Understanding Society dataset
its: a subfolder	etsim1.R etc.	Additional IPF iterations
R: folder containing the R code to integerise the weights generated through IPF	int-meth1-round.R	Simple rounding method
	int-meth2-thresh.R	Deterministic threshold method
	int-meth3-cw.R	Counter-weight method
	int-meth4-pp.R	Proportional probability method
	int-meth4-pp-many-runs.R	Many runs of PP method
	int-meth5-TRS.R	Truncate, replicate sample method
	int-meth5-TRS-many-runs.R	Many runs of TRS
	Analysis.R	Analysis of the results of integerisation
	outputs.R	Code to generate some results comparing the 3 integerisation methods
	Plotting-ints.R	Plotting commands
	Iteration-20.RData	Results of the 20th iteration of IPF
OA-eg	see section 4	Adapted model (for alternative inputs)

To use the data, the first stage is to set the working directory. Find out the current working directory using the command `getwd()` from the R command line. Correctly setting the working directory will allow quick access the files of the microsimulation model and a logical place to save the results. The command `list.files()` is used to check the contents of the working directory from within R. Assuming the folder 'ints-public' has been extracted to the desktop in a Windows 7 computer with the user-name 'username', type the following into the R command line interface and press enter to set the working directory (change 'username' to your personal user name or retype the path completely if the folder was extracted elsewhere):

```
setwd("C:/Users/username/Desktop/ints-public/etsim")
```

To run the model, one can simply type the following (warning: this may take several minutes, so entering the code block-by-block is recommended):

```
source("etsim.R")
```

If the aim is to understand how the method works, we recommend opening the script files using a text editor and sending the commands to R block by block. This can be done by copying and pasting blocks of code into the R command line. Alternatively a graphical user interface such as Rstudio can be used. In both cases, running the code contained in `etsim.R` should take around one minute on modern computers, depending on the CPU. This will result in a number of objects being loaded onto your R session's workspace. These objects are listed by the command

```
ls()
```

and can be referred to by name. The constraint variables, for example, can be summarised using the command:

```
 summary(all.msim)
```

R objects can also be loaded directly, having been saved from previous sessions. The command:

```
 load("iteration-20.RData")
```

for example, when run in the working directory 'R', will load the results of the IPF model results after 20 iterations. This may be useful for users who want to move straight to integerisation, without running the IPF model first. Referring to file-names in R can be made easier using the auto-complete capabilities of some R editors. Rstudio, for example, allows auto-completion of file-names and R objects (see RStudio's website for more information).

2 Running the spatial microsimulation model

The code for running the spatial microsimulation model is contained within 'etsim.R' the folder entitled 'etsim' — see Table 1. As with all R script files, the contents of this file can be viewed using any text editor. With an R console running, R's reaction to each chunk of code can be seen by copy and pasting the script code line-by-line. This should give some indication of how the model works, and which parts take most time to process. Note that R accepts input from external files. Within 'etsim.R', this technique was used to reduce the number of lines of code and make the model modular. The constraint variables, for example, are read-in using the following command, that is contained within the main etsim.R script file:

```
source(file="cons.R")
```

As before, this simply sends the commands contained within the file to R, line by line, but without displaying the results until the script has finished.

It is good practice to provide comments within the code, so that others can see what is going on. In R, this is done using the hash symbol (`#`). Anything following the hash is ignored by R, until a new line is formed.

Once the first iteration of the entire model has run, you can check to see if the model has worked by analysing the objects that R has created. The raw weights are saved as 'weights0', 'weights1', etc. The number of each set of weights corresponds to the constraint which was applied. All of 'weights0' are set to 1 in the first iteration (the initial condition). The object 'weights5' represents the cumulative weight so far, after the weights have been constrained by all 4 constraint variables.

The simulated zonal aggregates are stored in objects labeled 'USd.agg' (this stands for 'Understanding Society dataset, aggregated'), from the original value (the summary results of the survey data) to 'USd.agg4' (after fitting for the fourth constraint). A good first indication of whether the model has worked is to compare 'USd.agg4' with 'all.msim' (the latter being the census aggregates). This can be done by using the command:

```
head(USd.agg4)
```

and running the same command for 'all.msim'. The command `head()` simply displays the first 5 rows or elements of an R object, to get a feel for what it looks like. (The meaning of any command can be prefixing the command name by "?". In this case `?head()` would be used.) To make the comparison more interesting, one can plot the results. Try the following:

```
plot(all.msim[,13],USd.agg4[,13])
```

The '`[,`' part of this command means "all rows within"; the '`13]`' part means "in column 13". In this model, column 13 is the variable "mainly works from home" (`"mfh"`). This can be established using `names(all.msim)`, to identify the variables contained within the dataframe. If the plot looks the same as as that illustrated below (Fig .1), the model has worked.

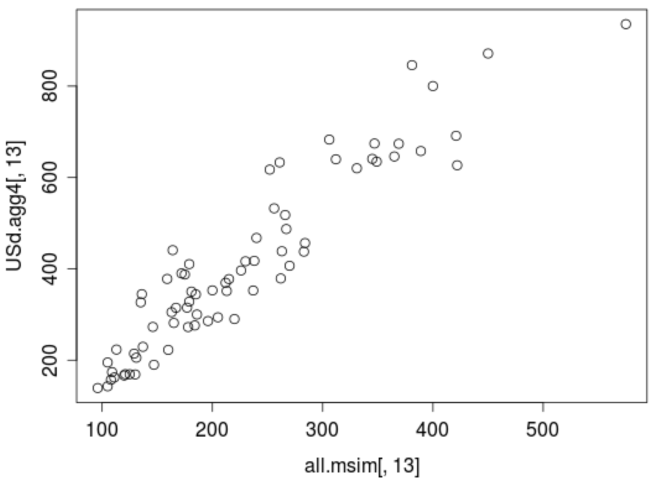

Figure 1: Diagnostic scatter plot to check if the model has worked.

3 Integerisation in R

The code used for integerisation of the results of IPF reweighting are kept in a separate folder, entitled 'R' (see Table 1). As before, navigate to this folder using a modified version of the following command, this time navigating to the folder 'R':

```
setwd("C:/Users/username/Desktop/ints-public/R")
```

As always, it is worth opening the script files in a text editor or within a dedicated R development environment such as RStudio. This will allow the commands to be seen in context and experimented with.

3.1 Simple rounding

The following section describes the code contained within the file 'int-meth1-round.R'. Open the file and send its content to R line by line. The aim of this script is to round the IPF weights (calculated in the previous section) up or down and then select individuals accordingly. Once the IPF weights have been loaded using the `load()` command, the new weights are created using the following command:[3]

```
intp <- round(i20.w5)
```

In the following line of code, the decimal remainders are saved by subtracting the rounded weights from the original weights. Note that each new set of data is given a name, ready to be referred back to later:

```
deci <- i20.w5 - intp # Decimal weights
```

Before running a loop to select individuals based on their rounded weights, we created a number of R objects to be used during integerisation. Of note, the object `pops` is a dataframe for saving data about the population of each zone that are calculated while the loop is in operation. It has the same number of rows as there are areas in the constraint table (`1:nrow(all.msim)`). The columns are bound together using `cbind()`. The contents of this object are updated with each iteration, allowing the results for different methods to be compared directly.

In order to perform calculations on one zone at a time, a loop is used:

```
for (i in 1:nrow(all.msim)){ ... }
```

The commands contained within the curly brackets are performed many times, once for each area. The index lists the row name of all individuals within the area `i` who have a rounded weight above 0 — `which(intp[,i]>0)`. The corresponding weight is referred to by `intp[which(intp[,i]>0),i]`.

[3]Here 'i20.w5' refers to the weights that emerge after the 4^th constraint of the 20^th iteration. Any weights can be used. For example 'i1.w5', if loaded into the R workspace, represents the weights after a single iteration of IPF.

The final list of individuals is saved by replicating the integer weights the same number of times as the integer value of the rounded weights:

```
ints[[i]] <- index[rep(1:nrow(index),index[,2])]
```

The replication command `rep()` is used in this instance to replicate the individuals who are 'cloned' more than once. Note the use of double square brackets. This is used to refer to objects (dataframes in this case) that are part of a list. Because the matrix of rounded IPF weights ('intp') has indexes that correspond to the original survey data, we can extract their characteristics by simply referring to the previously defined index:

```
intall[[i]] <- USd[ints[[i]],]
```

Finally, the results are aggregated by converting the raw data into the categories of the constraint variables — using `source("area.cat.R")`— and then summing columns to provide zone-wide counts for each category:

```
 intagg[i,]   <- colSums(area.cat)
```

This same procedure is followed for each of the remaining 2 integerisation methods. The defining features of each are outlined below.

3.2 The inclusion threshold approach

The starting point of this method is an incomplete simulated population of integer results (the length of which is defined as Pop_{sim}). The 5 steps of the threshold approach are as follows:

1. Set the initial value of the inclusion threshold IT to 1.
2. If the simulated population is too small ($Pop_{sim} < Pop_{cens}$), run the following loop (if not skip it).
3. Re-sample or 'clone' any individuals whose decimal weights[4] are less than IT yet greater than or equal to $IT - x$, where x is a small number to be iteratively subtracted from IT (Ballas et al. (2005) — in SimBritain: a spatial microsimulation approach to population dynamics — suggest $x = 0.001$; this value was also used here).

[4]By 'decimal weight', we refer to the value of a weight to the right of the decimal point. So, for a weight of 1.8, the 'decimal weight' is 0.8. Mathematically, the decimal weight (which we also refer to as the 'weight remainder') can be defined as $w - trunc(w)$ where the function trunc() removes all information to the right of the decimal.

4. Recalculate Pop_{sim} with the additional individuals included.

5. Subtract x from IT to reduce the inclusion threshold for the next iteration. If Pop_{sim} is still less than Pop_{cens} return to step 2; if not exit.

The script file 'int-meth2-thresh.R' replicates these steps in two main loops, each iterating over the areas whose populations are being simulated. The first is identical to that of the simple rounding approach, (except in this case the IPF weights are truncated, not rounded)[5] and saves the resulting microdata as a list of vectors, each containing row names of individuals from the Understanding Society dataset (see `ints[[i]]` for area `i`).

The second loop adds additional individuals to those contained in `ints` for each area, by gradually reducing the inclusion threshold. This is done in a third loop which is nested within the second. Note that the value of the threshold (`wv`, for weighting variable — equivalent to IT, as described above) is set to 1 outside this third loop. This is done so that the threshold is reduced from one iteration to the next within the loop:

```
 wv <- wv - 0.001
```

Note also that this third loop is initiated by the command `while()`, instead of the command `for()` used for the previous loops. This is because the number of iterations performed by the first two loops is fixed (to the number of areas), while the number of iterations in this one is determined by the threshold at which the sample population is greater than or to equal the census population:

```
while (length(ints[[i]]) < pops[i,1]){
```

Within this sub-loop additional individuals are added whose weights are between `wv` and `wv`-0.001:

```
ints[[i]] <- c(ints[[i]], which(dr[,i] < wv & dr[,i] >= wv - 0.001))
```

[5]This differs from the original implementation in the original SimBritain paper by Ballas et al. (2005): they used rounded weights as the starting point. However, after trying both methods, we found that beginning with truncated integer results leads to far less error introduced during integerisation. This is because topping up after simple rounding would lead an individual with a weight of 2.99 to be replicated 4 times: three times during rounding and once more as the inclusion threshold dips below 0.99.

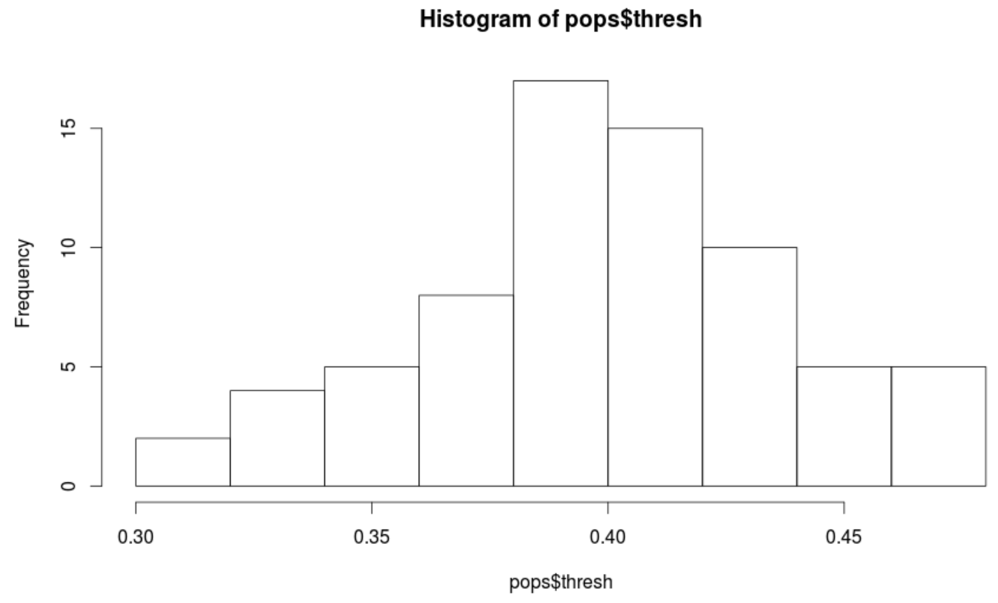

Figure 2: Histogram of the lowest value reached by IT (or 'wv' in the code) for all areas during the threshold approach.

Here, the command `c()` appends the additional individuals to those already saved. After the while loop exits, the population and aggregate data for each area are saved, as with the simple rounding method.

To analyse the threshold reached for each area, this information is saved as for each area within the main loop:

```
pops$thresh[i] <- wv
```

This information can be subsequently analysed, e.g. to investigate the distribution of thresholds reached (Fig. 2 — This plot was produced by the following command: `hist(pops$thresh)`). A similar process is used to save information about the exit point of the counter-weight algorithm.

3.3 The counter-weight method

The counter-weight method is similar to the threshold algorithm: it begins with a crude integerisation strategy (in this case simple rounding, not truncation as described above — this starting point was found to lead to more accurate results), and then tops up each area with additional individuals that depend on their decimal weights.

The process can be summarised in the following 4 steps:

- Sort the IPF weights in ascending order:

  ```
  sweights <- sort(i20.w5[,j], index.return = T)$x
  ```

  and save their order for future reference:

  ```
  ord <- rank(i20.w5[,j], ties.method="first")
  ```

- If the total population is too small, top up the results for each individual by the rounded sum of their decimal weight plus the decimal weight of the next individual in the sorted vector of weights:

  ```
    if(sum(iweights) < round(sum(sweights))){
        iweights[i] <- iweights[i] + round(dweights[i] + dweights[i+1])
        e[j] <- i
      }
  ```

- Update the integer weight vector for each area, including topped-up individuals, and re-order:

  ```
      intp[,j] <- iweights # but the order is wrong
    intp[,j] <- intp[ord,j]
  ```

- Convert these weight vectors into a list of individuals with replicated weights leading to replicated (cloned individuals):

  ```
    for(i in 1:j){
    index <- cbind((which(intp[,i]>0)) # generates index
                  ,intp[which(intp[,i]>0),i]) # integers)
  ```

```
  ints[[i]] <- index[rep(1:nrow(index),index[,2])] #clone
  pops$pcounter[i] <-  length(ints[[i]]) # save integer dataindividuals
  intall[[i]] <- USd[ints[[i]],] # Pulls all other data from index
  source("area.cat.R")
  intagg[i,]   <- colSums(area.cat)
}
```

Finally, the aggregate results for this integerisation method are saved as with previous methods, in this case as intagg.cw:

```
intagg.cw <- intagg
```

3.4 Proportional probability method

The script file 'int-meth3-pp.R' also contains two loops. The first simply creates proportional weights for each individual-zone combination using the following command: `i20.w5[,i] / sum(i20.w5[,i])`. This is the code equivalent of the following equation:

$$p = \frac{w}{\sum W} \tag{1}$$

The result (saved as `prop.weights[,i]`) is used in the second loop as the selection probability for each individual.

The second loop contains three main parts. First, individuals are randomly selected from the USd dataset, with probability set as follows:

```
prob=prop.weights[,i]
```

(Note that here we are sample *with* replacement — `replace=T`). Second, the population of the integerised sample is saved. Third, as with all integerisation methods, the command `source("area.cat.R")` is run to extract the additional information about individual from the Understanding Society dataset, based solely on their index. The results are saved as `intagg.prop`. The next stage is to run the TRS integerisation method.

3.5 TRS integerisation in R

The final method is contained in the script file 'Int-meth4-TRS.R'. It involves weight truncation, replication of integerised weights, and sampling based on the decimal remainders. Of these steps, sampling is the only one which requires detailed attention here: the others have already been described. Suffice to say that integer weights are generated by the command `x%/%1`, which is synonymous with the command `trunc(x)`. Note that the command `round()` was used for integerisation in the simple rounding and threshold integerisation methods.

The population following truncation is guaranteed to be less than the census population as no rounding up occurs. This differs from the simple rounding and threshold approaches, and ensures that there will always be a difference between census and simulated results. The challenge is to fill the difference:

```
popstrs[i,1] - popstrs[i,2]
```

where `popstrs[,1]` is the census population and `popstrs[,2]` is the simulated population based on truncated weights. The command:

```
 sample()
```

allows an exact number of rows to be selected to make up the difference. The first argument of the command is the vector from which the sample is taken. The second is the sample size. For our purposes, the vector is the row names of all individuals from the survey. This vector is referred to by the command `which(i20.w5[,i]>-1)`, which means "all individuals with weights greater than -1, for area `i`", i.e. all individuals. The size is the difference between census and simulated population sizes for the area in question (as defined above).

So far so good, but the sample strategy is simple random, meaning that probabilities will be equally assigned to all rows, unless stated. This is where the decimal weights — the 'conventional weight' components of the IPF weights — come into play. Conventional weights can be used to determine the probability of an individual being selected.

The final argument used, therefore, is the probability of selection (`prob=...`). The decimal weights are calculated in-situ by subtracting the integer weights from the actual weights:

```
prob = i20.w5[,i]-i20.w5[,i] %/% 1))
```

As with the previous methods, the loop finishes by extracting the full survey data from the survey dataset, and saving the aggregate level results:

```
 intagg.trs[i,]   <- colSums(area.cat)
```

After the script files associated with all four integerisation methods have been run, the aggregate results are saved in R objects entitled `intagg.round`, `intagg.thresh` and `intagg.trs`. These results form the basis of the integerisation method performance comparison presented in the paper, and can be replicated using the file 'Analysis.R' (Table 1).

4 Adapting the model

So far the model has been used on a single case study. For the techniques showcased here to be truly useful, they must be be applicable to a wide range of situations. This section therefore illustrates how to adapt the model to simulate the individuals living in Output Areas (which contain around 300 people or ∼100 employed people, 20 times smaller than the Medium Super Output Areas used up until now), using different constraints and a different (smaller) survey dataset from which individuals are to be extracted.

4.1 Setting-up the constraint variables

In order to show the model's flexibility, 3 new constraint variables were used:

- Hours worked per week
- Marital status
- Housing tenure of home

These variables are available in both aggregate form for small areas, and from the Understanding Society dataset. The aggregate data can be downloaded by UK academics from the Casweb census data portal. The raw data (named 'hrs_worked.csv' 'marital_status.csv' and 'tenancy.csv') is read into R and cleaned by the commands contained in the script file 'cons.R' within the folder 'OA-eg'. The comments in this script file should explain most of the commands, which read the .csv files and remove superfluous variables. In one case (tenancy) the variables are also manipulated such that the category ‘other’ is the sum of three other variables:

```
ten$other <- ten$other + ten$council + ten$assoc
```

The reason for modifying the data in this way is so that the constraint data match the individual-level survey data. Also, the USd is a huge dataset (50994 rows by 1322 columns, contained in a 90 Mb file). Dropping unneeded information makes the data more manageable.

The script to load and subset the USd data is contained in the file ‘load.R’ (also in the folder ‘OA-eg’). For confidentiality reasons the original data is not provided; the steps taken to process the USd dataset into a form ready for spatial microsimulation should be applicable to any survey dataset (the R package ‘foreign’ may be used to load unusual data types as an R object). The steps taken here should be fairly self-explanatory, based on the names of the commands and the comments. Although the script has been set-up to process the USd survey, in anticipation of running IPF constrained by the three constraint variables mentioned above, it would be possible to modify ‘load.R’ to accept different input survey datasets and subset the data for different constraints.

The data is also simplified to match available constraint categories in ‘load.R’. To provide one example, the USd variable for married status — ‘pmarstat’ — contains 14 categories, many of which can be merged. To ensure the categories of the survey data matched the census constraints (5 marriage status categories), the following command was used:

```
levels(Und.sub$mas <- s[sample(nrow(s), size=500),]rstat) <- c(
    rep("other",5), "single", "married", "single",
         "separated", "divorced", "widowed", rep("other",3))
```

After running both ‘load.R’ and ‘cons.R’ we are left with four R objects in the workspace:[6] ‘s’, the survey micro-level dataset and ‘hrs’, ‘mar’ and ‘ten’ — the three constraint variables.

4.2 Modifying the spatial microsimulation model

The script that runs the spatial microsimulation model in the previous example is called ‘etsim.R’. In order for it to use new constraint variables it must

[6]Due to data confidentiality, the full USd dataset cannot be provided. However, the data that results from ‘load.R’ has been saved as ‘oa-data.RData’ in the example folder.

be modified. These modifications (which maintain the original structure and semantics of the original script) can be seen by comparing ‘etsim.R’ contained within the ‘OA-eg’ folder against the file of the same name contained within the folder ‘etsim’. The following points summarise the changes made:

- Add or remove constraints and loading functions depending on the input data. In this case, for example, the input survey dataframe ‘s’ is too large relative to the average size of the zones under investigation (nrow(s) = 1678, more than 10 times greater the average size of individuals in Output areas — $\sim$ 100). Therefore a simple random sample is taken to reduce the number of rows to 500:

  ```
  s <- s[sample(nrow(s), size=500),]
  ```

- Alter the file ‘USd.cat.r’ so to convert the survey dataframe ‘s’ into a wide data frame whose dimensions match ‘all.msim’. This involves converting categorical variables into binary (1 or 0) using the subsets. Females who work more than 48 hours per week, for example, are allocated the value of 1 in the appropriate column using the following command:

  ```
  s.cat[which(s$jbhrs >= 49 & s$sex=="female"),12] <- 1
  ```

- Names of the R objects referred to are changed to reflect the new input data. The object ‘USd’, for example, is renamed as ‘s’.

4.3 Integerisation of the new results

The integerisation scripts must also be modified slightly to accept the new input data. Therefore the files ‘int-meth1-round.R’ to ‘int-meth4-TRS.R’ described in Table 1 have been altered. The changes we need to account for include the new name of the weights (i1.w4 instead of i20.w5 in this case — only iteration of the new model has been run for brevity) and, again, the renaming of the survey to ‘s’ from ‘USd’ in the original files. It is recommended that differences in the R scripts for integerisation between files in the folder ‘R’ and those (with the same file names) in the folder ‘OA-eg’ are identified to understand how the methods can be generalised to accept any weighted input data.

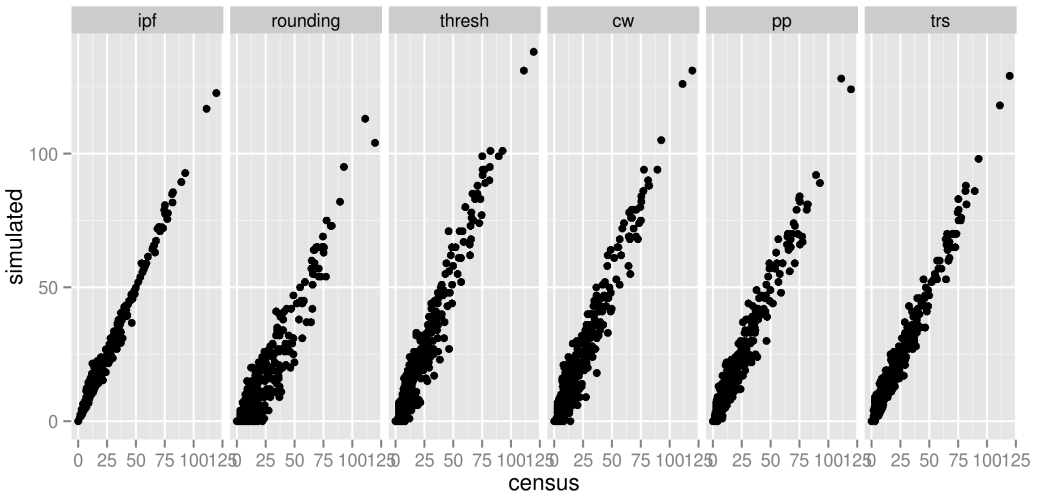

Figure 3: Scatter plots illustrating the relationship between census constraint variables (x axis) and simulated counts for these variables (y axis) after IPF and four methods of integerising the results. Each dot represents one variable for one area, 528 dots in each plot (22 variables multiplied by 24 areas).

4.4 Results

To confirm that the TRS method advocated in the paper is also the most accurate when it is used on different input data, a basic analysis script has been compiled ('basic-analysis.R' with the folder 'OA-eg'). These commands calculate the correlation between the simulated and census data at the aggregate data and illustrate the results. The results demonstrate that the TRS method is also more accurate than the others for these new constraints, as expected. The level of correlation rises (from 0.948 through 0.976, 0.975, and 0.981 to 0.987) for the threshold, rounding, counter-weight, proportional probabilities and TRS methods respectively. Note, the order of accuracy is the same as the same as presented in paper which this Supplementary Information accompanies, except for the counter-weight method performs worse than the inclusion threshold approach with the new input datasets.

These results can be visualised in scatter plots of census vs simulated results (Fig. 3). This figure can be replicated using the last section of code in 'basic-analysis.R', provided the packages 'reshape2' and 'ggplot2' have been installed.

We encourage users to test the integerisation methods described in this user manual on a wider range of datasets, citing the authors where appropriate. This will help to check the replicability of the results presented in the paper that accompanies this code. It is also hoped that the code and the findings will be of use to researchers developing, evaluating and using spatial microsimulation models.

Any feedback would be gratefully received by robin.lovelace at shef.ac.uk. There is also the possibility to clone, branch and commit to a larger code development project related to this research: `https://github.com/Robinlovelace/IPF-performance-testing`.

5 Reference

Lovelace, R., & Ballas, D. (n.d.). "Truncate , replicate , sample": a method for creating integer weights for spatial microsimulation. Computers, Environment and Urban Systems. (In press).