Global Optimisation (GO) User Guide

Dr Paul Williamson & Dr Xin Gu

Dept. of Geography & Planning

Version: 17th May 2018

Acknowledgements

A prototype version of this GO R package was developed by Dr Ferran Espuny-Pujol, in collaboration with Dr Paul Williamson and Dr Karyn Morrissey, as part of the ESRC-funded project 'Small area estimation of comorbidity'.

The mathematical underpinnings of GO are elucidated in Espuny-Pujol, F., Morrissey, K., & Williamson, P. (2018). A global optimisation approach to range-restricted survey calibration, *Statistics and Computing,* 28, 427-439.

The current version of the GO package was developed by Dr Xin Gu, in collaboration with Dr Paul Williamson, as part of an ESRC/NCRM-funded project 'Innovations in Small Area Estimation'.

New features

The prototype version of GO has been updated to (a) minimize run times by dropping the use of sparse matrices unless really needed; (b) provide more user control; (c) offer the option of using an open license LP solver; (d) produce biased (quick) or unbiased (a bit slower) integer solutions. The revised version of GO also includes three new functions to produce (i) tailor-made reports on fit to benchmarks; (ii) calibrated survey estimates; (iii) a set of synthetic population microdata.

Global Optimisation (GO) User Guide

Description

GO is an R package that can be used to calibrate survey weights, such that the sum of weighted dummy survey covariates are equal to the corresponding area-level benchmark constraints, whilst minimizing the chi-square distance in change from initial weights. In contrast to other calibration solutions, GO guarantees a globally optimum solution, in the sense that the resulting solution will have the minimum possible Total Absolute Error for the problem in hand.

Methods

GO calibrates survey weights by minimizing the chi-square distance from initial weights and being subject to benchmark constraints in each of the areas of interest. This can be formalized as

Minimize $$\sum \frac{(w_i - w_0)^2}{w_0}$$

Subject to

$$\boldsymbol{wA} = \boldsymbol{B}$$

$$\sum \boldsymbol{w}_i = B_T$$

$$\boldsymbol{L} < \boldsymbol{w} < \boldsymbol{U}$$

where $\boldsymbol{w}$ is a vector of survey calibration weights, $\boldsymbol{w}_0$ is a vector of initial weights, $\boldsymbol{A}$ is the survey represented in matrix form, $\boldsymbol{B}$ is the vector of calibration benchmarks for one area, B_T is the target sum for each benchmark variable, and $\boldsymbol{L}$ and $\boldsymbol{U}$ are lower and upper bounds to the calibration weights, specified as vectors of constants.

Functions

The R package GO comprises a number of functions:

- GO() is the main function. It produces real and/or integer calibrated survey weights based on the supplied survey data and calibration benchmarks.
- `fit()` assesses how well the calibrated survey data compare to the target benchmark constraints
- `estimate()` applies the calibration weights produced by GO to the survey to produce weighted counts, proportions and summary statistics
- `synthetic.pop()` applies the calibration weights produced by GO to the survey to generate a set of synthetic population microdata

Full details of these functions follow. For details of the nature and format of the required input data see the description of the GO() function.

1. GO()

The GO( ) function takes the following parameters:

```
GO(formula, survey_data, benchmark, initial_w = NULL,
   integer = FALSE, unbiased = TRUE, bounds = NULL,
   area_bm = NULL, minTAE = TRUE, sf = NULL,
   path = NULL, optimizer = NULL, seed = 100)
```

1.1 REQUIRED ARGUMENTS

Required arguments have to be specified by the user, as they have no default value.

survey_data

Users are required to supply a set of survey data, either as an R data frame object; or as a readable "csv" or "txt" file.

If the calibration benchmarks involve any person-level constraints, then for each person in the survey the supplied survey data must include:

- a unique person ID
- a household ID (but only if the benchmarks also include a household level constraint)
- a value for each calibration variable
- (optionally) the initial weights of the survey

If the calibration benchmarks involve only household level constraints, then for each household in the survey the supplied survey data must include:

- a unique household ID
- a value for each calibration variable
- (optionally) the initial weights of the survey

Table 1 illustrates this format for a set of survey data including person and household level variables

Table 1

Person	Household	N.Cars	Age	Sex	w0
1	1	2plus	65plus	F	0.3
2	1	2plus	16_49	F	0.4
3	2	1	50_64	M	0.5
4	2	1	50_64	F	0.5
5	2	1	0_15	M	0.3

In the above table `N.Cars` is a household-level variable recording the number of cars in each household, whilst `Age` and `Sex` are person-level variables.

The only constraints on naming columns and variable categories are that:

(1) Columns storing the unique Persons and Household IDs must be headed 'Person' and 'Household' respectively
(2) The column storing the initial weights (if any) must match the name specified in the function parameter `initial_w`

(3) All column names must begin with a letter, an underscore (_) or a period (.); and may contain only letters, numbers, underscores and periods
(4) Category values labels may be numeric, or a text string comprising only letters, numbers, underscores and periods.
(5) The values of the weights for each survey unit (if supplied) must be numeric

Table 2 shows an alternative form of Table 1, in which each category is represented by numeric value rather than by a text string:

Table 2

Person	Household	N.Cars	Age	Sex	w0
1	1	2	4	2	0.3
2	1	2	2	2	0.4
3	2	1	3	1	0.5
4	2	1	3	2	0.5
5	2	1	1	1	0.3

Table 3 documents the mapping of the numeric category labels from Table 2 onto the categories used in Table 1.

Table 3

Age		Sex		N.Cars	
Value	Categories	Value	Categories	Value	Categories
1	0-15	1	M	1	1
2	16-49	2	F	2	2plus
3	54-64				
4	65plus				

Note: The variable and category name are used in combination by GO to uniquely identify each variable category in the survey – E.g. `Age.0_15`; `Sex.2` etc.

`benchmark`

Users are required to supply a set of benchmark data via the `benchmark` parameter. These benchmark data are used as a set of area-level constraints on the survey calibration process.

The benchmark data may be supplied as a data frame object; or as a readable "csv" or "txt" file.

For each calibration area the benchmark data must contain:

- A column headed 'Area' which records the unique area name or ID
- A column headed 'Pop' which records the total number of persons in the area (or the total number of households if **any** of the benchmark constraints are at household level)
- One column per benchmark constraint (variable category)

By way of example, Table 4 reports a set of benchmark data suitable for accompanying the survey data illustrated in Table 3:

Table 4

Area	Pop	Age.0_15	Age.16_49	Age.50_64	Age.65plus	Sex.M	Sex.F
Area.1	200	30	80	70	20	100	90
Area.2	300	90	100	60	50	130	170
Area.3	200	65	40	45	50	80	120
Area.4	200	40	50	40	70	90	110

Note that for the benchmark data:

- One benchmark constraint must be provided per category in the relevant survey variable.

  For example, the survey data in Table 1 include an age variable with four categories. The benchmark data in Table 4 therefore include four age-specific benchmark constraints that precisely map onto these survey variable age categories.

- Each benchmark constraint must be named using the relevant variable and category labels from the accompanying survey dataset, with the variable and category labels separated by a period.

  For example, Age (variable name) + 0_15 (category name) ➔ Age.0_15

- The set of benchmark constraints provided for a benchmark variable do not have to sum to the area-level total ('Pop'); nor do all benchmark variables within the same area have to sum to the same amount.

  For example, in Area.1 the target population, 'Pop', is 200. The sum of the age constraints for Area.1 matches this target (30 + 80 + 80 + 70 = 200), but the sum of the two sex categories does not (100 + 90 = 190).

- If a variable's benchmark constraints do not sum to the target area-level population, 'Pop', they will not be fully satisfied by the post-calibration weights. This is because the post-calibration weights are guaranteed to sum to the 'Pop' value.

  For example, in Table 4 above the Area.1 benchmark target of 100 males and 90 females will not be fully met by the post-calibration weights, since the calibration weights will include an additional 10 persons to satisfy the target area population ('Pop') of 200.

Table 5 gives an example of a set of benchmark data involving a household-level constraints:

Table 5

Area	Pop	N.Cars.0	N.Cars.1	N.Cars.2plus	Sex.M	Sex.F
Area.1	200	80	70	50	250	250
Area.2	300	110	120	70	310	290
Area.3	200	75	80	45	270	230
Area.4	200	60	85	55	240	260

Since the benchmark data now include a household-level variable (`N.Cars`), 'Pop' now records the total number of households in the area, rather than the total number of persons. After calibration the categories of the household-level variables will sum to this total (200 for `Area.1`). In contrast, the sum of the categories within the Person-level variable `Sex` (300 for `Area.1`) will not. During the calibration process this discrepancy is resolved by treating the total number of households as a

'hard' constraint (no error allowed); and the total number of persons as a 'soft' constraint (error allowed). It is not possible to treat both the number of households and the number of persons as hard constraints, as this can cause convergence problems during the calibration process. However GO will find the minimum error solution achievable for a given set of survey data and benchmark constraints.

For the sake of simplicity, here the focus has been only on univariate benchmarks with whole population coverage. See Section 1.6 for further information on how to handle benchmarks that do not provide full population coverage (e.g. working-age population only); or that involve an *n*-way interaction between benchmark variables (e.g. a two-way *Age* by *Sex* benchmark).

`formula`

If no formula is provided, then by default GO assumes that every single variable category in the survey has a matching constraint in the benchmark data, and that the variables in the survey should be at the same level.

If this is not the case (because not all of the survey variables are to be used as calibration constraints), then users are required to provide a formula that specifies which survey variables are to be used as calibration constraints.

For *person-level constraints*, the formula required is of the form

```
Person ~ Var1 + Var2
```

where `Var1` and `Var2` are person-level benchmark variables

For *household-level constraints*, the formula required is of the form

```
Household ~ Var1 + Var2
```

If the benchmarks include a *mix of person-level and household-level variables*, then a list of formulas is required:

```
list( Person ~ Var1 + Var2, Household ~ Var3 + Var4)
```

where `Var1` and `Var2` are person-level benchmark variables and `Var3` and `Var4` are household-level benchmark variables.

For example, an appropriate formula to represent the following set of benchmark constraints…

Area	Pop	**N.Cars** .1	**N.Cars** .2plus	**Age** .0_15	**Age** .16_49	**Age** .50_64	**Age** .65plus	**Sex** .M	**Sex** .F
Area.1	200	80	120	30	80	70	20	250	250
Area.2	300	110	190	90	100	60	50	310	290
Area.3	200	75	125	65	40	45	50	270	230
Area.4	200	60	140	40	50	40	70	240	260

…is:

```
list( Person ~ Age + Sex, Household ~ N.Cars )
```

where `Age` and `Sex` are person-level variables and `N.Cars` is a household-level variable.

1.2 *OPTIONAL* ARGUMENTS

GO also offers a number of optional input arguments to allow users to have greater control over the specification of their calibration problem. All optional input arguments have default values.

`area_bm`

This argument allows the user to specify which of the areas in the supplied set of benchmark areas is to be used during the calibration process.

This argument can be invoked in a number of ways:

area_bm = 4	Numeric selection; in this case of the fourth area in the set of supplied benchmark data
area_bm = c(1:2, 4)	Numeric vector selection; in this case selection of the first, second and fourth areas
area_bm = c("Area.1", "Area.2")	Selection of areas via a user-supplied set of one or more unique area IDs

The default value for this argument is NULL, which triggers the assumption that all areas in the benchmark will be used in the calibration.

`bounds`

This argument can be used to specify the lower and upper bounds of the calibrated weights. It is invoked by passing a vector with length two specifying the required lower and upper bounds.

For example, `bounds = c(0, 5)` specifies a lower bound of 0 and an upper bound on calibration weights of 5.

The default value for this argument is `NULL`, which restricts GO to using only positive weights, equivalent to invoking `bounds = c(0, Inf)`.

`initial_w`

This argument is used to declare the initial weights of persons (or households) in the survey being calibrated.

The default value for `initial_w` is `NULL`, which triggers the assumption that all persons in the survey have an equal weight (of 1).

To specify an alternative set of initial weights, three approaches are possible:

initial_w = "*column_name*"	Point to the name of the column in the imported survey data that stores the weights. E.g. initial_w = "w0"
initial_w = *vector_object*	Points to a pre-existing R vector object which stores the initial weights in the same order as survey units are listed in the survey dataset
initial_w = *formula*	Point to the name of the column in a formula type. E.g. initial_w = ~ w0

integer

This argument has three possible states.

integer = FALSE	The default value, which causes GO to report only the real weighted solution
integer = TRUE	Causes GO to report only the integer weighted solution
integer = "BOTH"	Causes GO to report both the real and integer weighted solutions

See Appendix 1 for details of the integerization process.

minTAE

This argument has two possible states

minTAE = TRUE	The default value, which causes GO to find a solution which minimizes the Total Absolute Error for the problem in hand
minTAE = FALSE	Causes GO to find a solution which minimises the Standardised Absolute Error (TAE divided by relevant benchmark table total)

Note: Counter-intuitively the same results are obtained whether minimizing TAE or SAE. The options are left for users to demonstrate the calibration procedure themselves.

sf

Allows the user to apply scaling factors to the benchmarks. The purpose of scaling is to (de)prioritise the fitting of specific benchmarks. The default is to apply a scaling factor of 1 to all benchmarks. If the corresponding benchmark variable/category is underestimated a scaling factor > 1 will cause a reduction in TAE. If the corresponding benchmark variable/category is overestimated then a scaling factor < 1 is needed to produce a reduction in TAE.

This argument can be invoked in the following ways:

sf = c("Age = 2", "Sex = 3")	Add scaling factors to specified benchmark variables. E.g., "Age" and "Sex".
sf = c("Age.0_15 = 3", "Sex.F = 2")	Add scaling factors to specified benchmark cells. E.g., "Age.0_15" and "Sex.F"
sf = c("Sex = 2", "Age.0_15=3")	Add scaling factors to a combination of benchmark variables and cells. E.g., "Sex" and "Age.0_15"

Note: GO will find a calibration weights that provided a minimum TAE solution to the scaled benchmarks, although scaling causes the minimum achievable TAE to change.

optimizer

This argument allows the user to specify which optimizer should be used to solve the calibration problem. The three options are:

optimizer = “limSolve”

optimizer = “Gurobi”

optimizer = “Rmosek”

The default option is `limsolve.`

The advantages and disadvantages of each optimizer is explained in Section 1.3.

unbiased

The `unbiased` argument allows users to specify whether they want to generate biased or unbiased integer weights. It is active only if the argument `integer` = TRUE.

Conventional optimizer solutions deliver biased results when converting real- into integer-weighted solutions, because they have a pre-disposition to round to the nearest integer, rather than rounding with a probability equal to the fraction being rounded. The advantage of the biased solution is that it is deterministic (same solution every run). Alternatively, GO offers the option to generate an unbiased solution (rounding with probability equal to the fraction being rounded). This unbiased solution is stochastic, meaning that multiple runs might generate multiple solutions. To generate multiple solutions, users need to supply a seed value via the `seed` argument.

unbiased = TRUE — The default value for this argument, which causes an unbiased integerisation method to be used to generate integer weights.

unbiased = FALSE — If FALSE, a biased binary search method will be used.

Note that with `optimizer = “limSolve”` only unbiased integer weights can be produced, and thus unbiased should be `TRUE` in this case.

For full details of the integerization process, including the distinction between biased and unbiased rounding, and the circumstances under which an ‘unbiased’ solution will actually be ‘minimally biased’, see Appendix 1.

path

This argument allows the user to specify which folder the survey and benchmark data need to be read from. (Note that both files need to be stored in the same folder.)

E.g.

path = “/Users/User1/Documents/GO/Data” [Mac\Linux]

path = "C:/Users/Name/GO/Data” [Windows PC]

The default value is NULL, which assumes that the current working directory is the path.

seed
This argument is active only if the arguments `integer` and `unbiased` are both set to `TRUE`. It takes an integer value that specifies the seed for unbiased integerisation. (For details, see the argument `unbiased`).

seed = 100 The default `seed` value

1.3 OPTIMIZER CHOICE

GO uses a mixture of linear and quadratic programming to calibrate survey weights.

- Linear programming is used to compute the minimum achievable total absolute error

- Quadratic programming is used to minimize chi-square difference (quadratic) in change from initial weights, subject to the constraints, which include the minimum achievable TAE.

The quadratic programming element requires the use of an optimizer especially designed to solve such problems.

GO offers the choice of three `R` packages to solve quadratic programming problems. The choice of optimizer can be specified via the `optimizer` argument:

optimizer = "limsolve"

optimizer = "Gurobi"

optimizer = "limSolve"

If no option is specified, the default optimizer used by GO is "limsolve"

Optimizer	*Advantages*	*Disadvantages*
limSolve[1]	Free to all	• Limited by maximum survey size (variables x survey units) • Slow if the survey size is relative large • Unable to provide biased integer weights
Gurobi	• Free to academics • No constraint on maximum survey size • Fast	Commercial license required by other users
Mosek	• Free to academics • No constraint on maximum survey size • Fast	Commercial license required by other users

[1] Uses the R package `limSolve` to find a real-weighted solution; the R-package `lpSolve` to find an unbiased integer solution; but is unable to find a biased integer solution because as implemented by GO this involves integer quadratic programming - something which neither `limSolve` nor `lpSolve` currently support.

Installing limSolve
Install the 'limSolve' and 'lpSolve' packages from the CRAN website

Installing Gurobi
Users will need to install the R package 'gurobi' from the Gurobi website. This package provides an interface to the Gurobi software.

They will also need to download Gurobi itself from
http://www.gurobi.com/download/gurobi-optimizer.

For full instructions on how to install gurobi for R, see:
https://cran.r-project.org/web/packages/prioritizr/vignettes/gurobi_installation.html and
http://www.gurobi.com/documentation/7.5/quickstart_mac/r_installing_the_r_package.html

Installing Mosek
Users will also need to install the R package 'Rmosek' via the CRAN website. This package provides an interface to the mosek software.

They will also need to download mosek from https://www.mosek.com/downloads/ .

For full instructions on how to install Mosek for R, see:
https://docs.mosek.com/8.1/rmosek/install-interface.html

1.4 GO OUTPUT

The output from the GO function is returned as a list of data objects. These objects are described in turn below.

`real_weights`

A matrix that contains the non-integer calibration weights. The number of rows equals the number of areas in the benchmark table (or length of `area.bm` if specified). The number of columns equals the number of persons (or the number of households if household level benchmark variables are included in the calibration). This output is only produced if the input argument `integer = FALSE` or `"BOTH"`.

`integer_weights`

A matrix that contains the integer calibration weights. It has the same dimensions as *real_weights.* This output is only returned if the input argument *integer* = TRUE or "BOTH".

`initial_weights`

A vector storing a copy of the initial weights used in the calibration (one weight per survey unit).

`survey_data`

A data frame containing a copy of the survey data being calibrated, including all survey variables, not just those used as benchmarks. Used by the `estimate()` function.

`survey_dummy`

A data frame containing one dummy variable per category in each calibration benchmark variable. Used by the `fit()` function.

benchmark
A data frame containing a copy of the benchmarks used in the calibration

num_cat
The number of categories within each of the benchmark variables used during the calibration (stored as a table)

area
A vector recording a list of the areas from the benchmark data used during the calibration, identified via their area code.

area.size
A vector recording the total population (or number of households) of each area used during the calibration

TAE
Total absolute error of the calibration problem. See details of the `fit()` function for details.

status
A report on the status of the calibration process.

1.5 EXAMPLE R CODE

The GO package comes pre-supplied with the two example datasets: a survey and a set of area-level benchmarks.

```
## Load package (including example datasets)
Library(GO)

## Examine example datasets
head(GO.survey_data)
head(GO.benchmark)

## Run GO and save results as a data object
res <- GO(formula= ~ Age + Sex,
          survey_data = GO.survey_data,
          benchmark = GO.benchmark,
          initial_w = w0)

## Run GO for first three areas only, returning integer weighted solution
res <- GO(formula = ~ Age + Sex,
          survey_data = GO.survey_data,
          benchmark = GO.benchmark,
          initial_w = w0,
          integer = TRUE,
          area.bm = 1:3)
```

See Appendix 2 for the R code used to create the example survey and benchmark datasets.

1.6 HANDLING MORE COMPLEX BENCHMARKS

Benchmark categories that do not cover whole population

Post-calibration weights are guaranteed to sum to the population total specified in the benchmark data column 'Pop'. This causes problems when the set of benchmark constraints for a variable do not cover the same population as that covered by 'Pop'. For example, benchmark constraints for occupational status might be available only for the working age population, whilst 'Pop' covers the whole population. As a result, the post-calibration sum of persons by occupational status category would be inflated to the 'whole population' target specified by 'Pop', causing the number of working-age persons in each occupation category to be grossly over-estimated.

In such cases it is necessary to:

(i) Add an additional benchmark category to capture the population omitted from the benchmarks

(ii) Create a revised version of the equivalent survey variable, which accurately maps onto this additional benchmark category

An example is provided in Tables 6 and 7.

Table 6, columns 1-4, shows the original benchmark data for the variable `occupation`, which only covers the working age population. An additional benchmark (final column) has therefore been added which provides coverage of the omitted non-working age population. The number of persons in this additional category was calculated as the total area population less the sum of the existing benchmark values. For `Area.1` this gives 200 – (50 + 50) = 100.

Table 6

Area	Pop	Occupation. Professional	Occupation. Manual	Occupation. Not_ WorkingAge
Area.1	200	50	50	100
Area.2	300	100	70	130
Area.3	200	75	45	80
Area.4	200	100	100	0

Table 7 shows the original survey data (first two columns) and, in the final column, the revised version of the variable required to map on to the amended set of benchmarks in Table 6.

Table 7

Person	Occupation	Occupation
1	Professional	Professional
2	Manual	Manual
3	School Pupil	Not_WorkingAge
4	Retired	Not_WorkingAge
5	Manual	Manual

Note that the name of the adjusted survey variable and the names of the revised categories must match the variable and category names used in the benchmark data.

N-way benchmarks

For the sake of simplicity, all of the examples provided so far have involved only 'univariate' benchmarks: i.e., benchmarks involving only one variable, such as the age distribution, or the occupation distribution. However, wherever possible, it is normally desirable to include benchmarks that capture the interactions between benchmark variables.

Table 8 illustrates a 2-way benchmark constraint for each area (`Age` by `Sex` counts).

Table 8

Area.1		Sex		Area Total
		Male	Female	
Age	0_15	25	20	
	16_49	30	25	
	50_64	30	25	
	65plus	15	30	**200**
Area.2		Sex		
		Male	Female	
Age	0_15	25	30	
	16_49	30	35	
	50_64	40	45	
	65plus	40	55	**300**

Table 9 illustrates the same data reformatted into the input format required by GO

Table 9

Area	Pop	Age.Sex. M.0_15	Age.Sex. M.16_49	Age.Sex. M.50_64	Age.Sex. M.65plus	Age.Sex. F.0_15	Age.Sex. F.16_49	Age.Sex. F.50_64	Age.Sex. F.65plus
Area.1	200	25	30	30	15	20	25	25	30
Area.2	300	25	30	40	40	30	35	45	55

Finally, Table 10 illustrates the original `Age` and `Sex` variables, plus the revised '`Age.Sex`' variable required to map onto this two-way benchmark data:

Table 10

Person	Age	Sex	Age.Sex
1	65plus	F	F.65plus
2	16_49	F	F.16_49
3	50_64	M	M.50_64
4	50_64	F	F.50_64
5	0_15	M	M.0_15

2. fit()

The *fit()* function assesses fit by comparing the benchmark targets with their calibration weighted equivalents. It returns a list comprising a series of data frames that report various aspects of the fit of the results from GO function to the benchmark targets.

If calibration result contains both integer and real calibration weights, then `fit()` additionally reports the same measures of fit for each type of calibration weight.

2.1 ARGUMENTS

The function `fit()` takes the following arguments:

fit(res, bm_areas="All", bm_vars="All", var_measures = "All", cat_measures = c("TAE", "Z"), dp=2)

These arguments are introduced in turn below.

res

The calibration results from GO

bm_areas

Allows the user to specify which of the areas in the set of calibration outputs are to be used when assessing fit. The default is to report the overall fit, plus the fit by area.

This argument can be invoked in a number of ways:

bm_areas = "All"	Reports the fit for each benchmark area, plus the overall fit across all areas. (This is the default setting.)
bm_areas = 4	Numeric selection. Reports the fit of the selected area (in this case the fit of the fourth in the set of calibrated areas), plus the overall fit across the selected areas. (The area-specific and overall fit are identical when only one area is selected.)
bm_areas = c(1:2, 4)	Numeric vector selection. Reports the fit of each selected area (in this case the first, second and fourth calibrated areas), plus the overall fit across the set of selected areas.
Bm_areas = c("Area.1", "Area.2")	Character string selection. Reports the fit of each selected area (specified as a list of one or more unique area IDs), plus the overall fit across the set of selected areas.

bm_vars

Allows the user to specify which of the benchmark variables are to be used when assessing calibration fit. The default is to use all benchmark variables.

The argument **bm_vars** can be invoked in a number of ways:

bm_vars = "All"	Causes all benchmark variables to be used (This is the default setting.)
bm_vars = 4	Numeric selection; in this case selection of the fourth in the set of benchmark variables
bm_vars = c(1:2, 4)	Numeric vector selection; in this case selection of the first, second and fourth benchmark variables
bm_vars = c("Age", "Sex")	Selection of benchmark variables via a user-supplied set of one or more variable names

var_measures

This argument allows the user to specify which measures of fit are to be reported for each benchmark variable. The default is to report all measures. See section on 'Measures' below for full details.

The argument options are:

var_measures = "All"	Returns TAE, MAE, PM, MPM, Zsq, RZsq, Zmsq, RMSE, RSD, r, rsq, slope and chi_wd.
var_measures = "*measure*"	Where *measure* is one of TAE, MAE, PM, MPM, Zsq, RZsq, Zmsq, RMSE, RSD, r, rsq, slope and chi_wd.
var_measures = c("TAE", "r")	String vector. User-supplied list of measures, chosen from TAE, MAE, PM, MPM, Zsq, RZsq, Zmsq, RMSE, RSD, r, rsq, slope and chi_wd.

cat_measures

This argument allows the user to specify which measures of fit are to be reported for each benchmark *category*. The default is *cat_measures = c("TAE", "Z")* which reports TAE and Z only. The section on 'Measures' below provides full details of these and the other measures of fit available.

Measures

Notation:

Aw = calibrated estimate, *B* = Benchmark target, *i* = i^{th} cell in in table, *n* = number of cells in benchmark table (variable), N_{Aw} = total calibrated population in table; N_B = total benchmark population in table.
The available optional measures of fit are:

1. TAE-based measures

Focuses upon absolute differences in *counts*

TAE — Total Absolute Error

Cell: $|Aw_i - B_i|$

Table: $\sum_i^n |Aw_i - B_i|$

MAE — Mean Absolute Proportional Error
[Standardises TAE for number of cells]

Cell: $|Aw_i - B_i|$

Table: $\frac{\sum_i^n |Aw_i - B_i|}{n}$

PM — Proportion Misclassified
i.e. % of units that need to change table cells to match benchmarks.
[Standardises TAE for size of population in table]

Cell: $\frac{TAE_i}{2N_B}$

Table: $\frac{\sum_i^n TAE_i}{2N_B}$

MPM — Mean Proportion Misclassified
[Standardises PM for number of cells in table]

Cell: Not an option ($MPM_i = PM_i$)

Table: $\frac{PM}{n}$

2. Z-score based measures

Focuses upon relative difference in *proportions*. See Voas and Williamson (2001) for full details.

Z

Z-score
Error in fit of proportions divided by standard deviation of that error.
[Standardises for table population]

Cell: $\dfrac{\frac{Aw_i}{N_{Aw}} - \frac{B_i}{N_B}}{\sqrt{\frac{\frac{B_i}{N_B}\left(1-\frac{B_i}{N_B}\right)}{N_{Aw}}}}$ when $N_B > 0$

Aw_i when $N_B = 0$

Table: No summary measure available

Zsq

Z-score squared
Follows a chi-square distribution with df = number of cells.

Cell: $Z_i{}^2$

Table: $\sum_i^n Z_i{}^2$

RZsq

Relative Z-score squared
Divides by critical value of chi-squared when significance level = 0.05 and *df* = *n*
[Standardises Z for number of cells in table]

Cell: $\dfrac{Zsq_i}{\chi^2(sig{=}0.05;\, df{=}1)}$

Table: $\dfrac{Zsq}{\chi^2(sig{=}0.05;\, df{=}n)}$

Zm

Modified Z-score
Zm = Z when $N_{Aw} = N_B$. When N_{Aw} and N_B differ, Zm places more emphasis on absolute differences than Z.

Cell: $\dfrac{\frac{Aw_i}{N_B} - \frac{B_i}{N_B}}{\sqrt{\frac{\frac{B_i}{N_B}\left(1-\frac{B_i}{N_B}\right)}{N_B}}}$ when $N_B > 0$

Aw_i when $N_B = 0$

Table: No summary measure available

Zmsq

Modified Z-score squared
Zmsq = Zsq when $N_{Aw} = N_B$. When N_{Aw} and N_B differ, Zmsq places more emphasis on absolute differences than Z

Cell: $Zm_i{}^2$

Table: $\sum_i^n Zm_i{}^2$

RZmsq

Relative Zm-score squared
Not calculated because RZmsq ≈ RZsq when $N_{Aw} \approx N_B$; and when N_{Aw} and N_B differ significantly, Zmsq has an unknown sampling distribution, hindering correct calculation of RZmsq.

3. Measures of variability in error

RMSE

Root Mean Squared Error
[Standardises for number of cells in table]

Cell: Not applicable

Table: $\sqrt{\sum_i^n \frac{(Aw_i - B_i)^2}{n}}$

RSD

Relative Standard Deviation
Standard Deviation as a % of mean absolute error
[Standardises for mean absolute error]

Cell: Not applicable

Table: $\frac{\sqrt{\sum_i^n \frac{(Aw_i - B_i)^2}{n-1}}}{\sum_i^n \frac{|Aw_i - B_i|}{n}} \; x \; 100$ where $n > 1$

4. Correlation and Regression measures

Note: These measures can be calculated for specific areas; specific variables; or for combination of the two. (E.g. Age in *Area.1*). The measures cannot be calculated if there is only one benchmark category in the specified combination. In these instances a value of NA will be returned instead.

r — Pearson's Product Moment Correlation

$$r = \frac{\sum_{i=1}^{n}(Aw_i - \overline{Aw})(B_i - \bar{B})}{\sqrt{\sum_{i=1}^{n}(Aw_i - \overline{Aw})^2}\sqrt{\sum_{i=1}^{n}(B_i - \bar{B})^2}}$$

where *n* = number of estimated benchmarks (no. of variable categories x no. of calibration areas for which fit is being assessed)

rsq — Coefficient of Determination

r^2

slope — Regression Slope
The slope of the line of best fit for the ordinary least squares regression model *Aw* = intercept + slope(*B*).

A slope of 1 indicates that, on average, the calibrated values neither over- nor under-estimate their benchmark counterparts

5. Measure of change from initial weights

chi_wd — The chi-square distance between the initial and calibration weights, reported for each calibration area specified via *bm_areas*. (See earlier Methods section for details of this measure.) Not applicable as a summary measure across all areas; hence only reported in output reporting fit by area.

dp

Controls the number of decimal places to which measures of fit are reported.

dp = 2 — Numeric value. (The default value is 2).

2.2 OUTPUT

The output from the *fit()* function is returned as a list of *data.frames* as follows:

(a) Fit by benchmark across all areas (benchmarks = all; variable; category)

```
$BM

BM            TAE    MAP    …    r   rsq   slope
All
Age
Sex
Age.0_15
…
Age.65plus
Sex.Male
Sex.Female
```

(b) Fit by area across all benchmarks

```
$Area

Area           TAE    MAP    …    r   rsq   slope   chi_wd
Area.1
Area.3
```

(c) Fit by benchmark variable by area

```
$Var

Area           TAE.Age   TAE.Sex   MAP.Age   …   rsq.Age   rsq.Sex …
Area.1
Area.3
```

(d) Fit by benchmark category by area

```
$Cat

Area        TAE.Age.0_15 … TAE.Sex.Female … rsq.Sex.Male   rsq.Sex.Female …
Area.1
Area.3
```

If also reporting results for integer weighted solutions, then the output will include a further set of data.frames, with the same layout as those already outlined, but with the data.fames named:

```
$BM.int
$Area.int
$Var.int
$Cat.int
```

This approach makes it very easy to find the difference between real and integer weighted fit.

E.g. `BM – BM.int`

3. estimate()

The `estimate()` function takes the following parameters:

estimate (res, target_vars, type, integer = FALSE, bm_areas = NULL)

`GO()` produces survey calibration weights. The `estimate()` function applies these calibration weights to the survey, to produce weighted counts, proportions and summary statistics for user-specified variables and benchmark areas. For example, the mean income per area; or the number or proportion of females per area.

3.1 ARGUMENTS

res
The calibration results from GO (which include the calibration weights).

target_vars
Specifies the survey variable(s) for which calibrated counts, proportions or summary statistics are required, using one or more formulae.

For *person-level variables*, the formula required is of the form

```
Person ~ Var1 + Var2
```

where `Var1` and `Var2` are person-level variables

For *household-level variables*, the formula required is of the form

```
Household ~ Var1 + Var2
```

If users are estimating a *mix of person-level and household-level variables,* then a list of formulas is required:

```
list( Person ~ Var1 + Var2, Household ~ Var3 + Var4)
```

where `Var1` and `Var2` are person-level variables and `Var3` and `Var4` are household-level variables.

type
For each user-specified variable supplied via `target_var,` identifies the data type. The options are “Cat” for categorical variables and “Con” for continuous variables. Data types should be identified in the same order that variables are supplied to the argument `target_var`.

E.g.

```
target_var = Person ~ Var1, type = “Cat”

target_var = Person ~ Var1 + Var2, type= c(“Cat”, “Con”)

target_var = list(Person ~ Var1, Household ~ Var2 + Var 3),
             type = c(“Cat”, “Con”, “Cat”)
```

integer

This argument has two possible states.

`integer = FALSE` The default value. Target variables are estimated using real calibration weights.

`integer = TRUE` Target variables are estimated using integer calibration weights.

Only works if `GO()` output contains the relevant type of calibration weights.

bm_areas

This argument allows the user to specify the benchmark areas for which estimates are required. See Section 1.2 of the description of the `GO()` function for details of how to invoke this argument.

3.2 OUTPUT

The `estimate()` function produces different types of results for categorical variables and continuous variables.

For categorical variables, the `estimate()` function reports a list of data frames (one per categorical variable specified via `target_vars`). Each data frame consists of the count of each category of the variable; the proportion of each category over all categories.

$Health

	Area	Health.good	Health.bad	Health.good.prop	Health.bad.prop
1	Area.1	100	100	0.50	0.50
2	Area.2	150	50	0.75	0.25

For continuous variables, *the* `estimate()` function reports a list of data frames (one per continuous variable specified via `target_var`). Each data frame consists of a statistical summary of the variable's distribution within each benchmark area. The summary values reported are:

Min	The minimum value. (For real calibration weights, interpret as the 0^{th} percentile)
Q1	1^{st} quartile (25^{th} percentile)
Median	Median (50^{th} percentile)
Mean	Arithmetic mean
Q3	3^{rd} quartile (75^{th} percentile)
Max	The maximum value. (For real calibration weights, interpret as the 100^{th} percentile.)
Gini	The weighted Gini coefficient $G = \frac{1}{n}\left(n + 1 - 2\frac{\sum_{i=1}^{n}(n+1-i)yi}{\sum_{i=1}^{n} yi}\right)$ for a set of n continuous values, y_i, where y_i is indexed in non-decreasing order ($y_i < y_{i+1}$).

$Income

	Area	Min	Q1	Median	Mean	Q3	Max	Gini
1	Area.1	0.062	82.479	153.495	169.898	251.878	447.369	0.751
2	Area.2	0.043	70.670	139.413	169.855	247.442	634.834	0.982
3	Area.3	0.069	80.627	151.209	169.865	237.123	609.448	0.347

4. synthetic.pop()

The function `synthetic.pop()` generates a synthetic population based on the calibrated integer weights from the `GO()` function. This is achieved by creating a copy of the calibrated survey, for each benchmark area, in which all persons (or households) with a weight of zero are dropped; each person (household) with a weight of one is retained as is; and each person (or household) with an integer weight of more than one is duplicated as many times as their integer weight requires. A field is added to each copy of the survey, recording the relevant area ID. The resulting set of survey copies (one per benchmark area) are then combined, creating a synthetic population which conforms as closely as possible to the originally supplied area-level calibration benchmarks.

The function has the form:

synthetic.pop(res, vars=NULL, bm_areas = NULL)

4.1 ARGUMENTS

`res`
Calibration results from GO

`vars`
Allows users to specify which variables from the survey supplied via *survey_data* are to be retained in the synthetic population. The default (*NULL*) is to retain all variables.

Note:

- Where the variables to be retained are user-specified, the resulting synthetic population will include variables in the order requested.

- In all cases the identifier variables *Area*, *Person* and, if relevant, *Household* from the source survey will be included as the first variables in the synthetic population. Hence these identifier variables do not need to be specified in the list of variables to be retained; and where specified, will be ignored.

- In addition the synthetic population will include a new 'synthetic' person / household ID. The synthetic IDs are required to distinguish duplicate persons and households in the same area from one another.

The argument `vars` can be invoked as follows:

vars = "Age"	Character string. Retains the identifier variables Area, Person and Household (where present), plus the variable *Age*.
vars = c("Age", "Sex", "Income")	Vector of character strings. Retains the identifier variables Area, Person and Household (where present), followed by the variables *Age, Sex* and *Income*.
vars = 10	Numeric. Retains the identifier variables Area, Person and Household (where present), followed by the tenth survey

variable (unless the tenth variable is one of the identifier variables already retained).

vars = c(1, 4:10) — Numeric vector. Retains the identifier variables Area, Person and Household (where present), followed by the first and fourth to tenth survey variables (excluding any of the first and fourth to tenth variables that are identifier variables).

`bm_areas`

This argument allows the user to specify the benchmark areas for which estimates are required. See the description of the `GO()` function, Section 1.2, for details of how to invoke this argument.

4.2 OUTPUT

The `synthetic.pop()` function generates synthetic population for the area specified. The output of the function is a data frame.

Example 1:

	Area	Person_syn	Person	Age	Sex	Health	Unemployed	Income
1	Area.1	1	3	3	1	2	1	164.2614
2	Area.1	2	7	2	1	3	1	156.4963
3	Area.1	3	12	1	2	1	-9	176.2714
4	Area.1	4	12	1	2	1	3	176.1616
5	Area.2	5	84	2	1	1	2	172.8943
6	Area.2	6	53	2	2	3	-9	167.3694

In the above example, in Area.1 Person 12 was assigned an integer calibration weight of 2, and thus is included twice in the resulting synthetic population. To flag this, the original and duplicate of Person 12 are assigned the same Person ID.

Example 2:

	Area	Household_syn	Household	Person_syn	Person	Age	Health
1	Area.1	1	6	1	13	1	1
2	Area.1	1	6	2	14	3	3
3	Area.1	1	6	3	15	2	1
4	Area.1	2	6	4	13	1	1
5	Area.1	2	6	5	14	3	3
6	Area.1	2	6	6	15	2	1

In the above example, Household 6 was assigned an integer calibration weight of 2. This household and persons in it are included twice. To flag this, the original and duplicate are assigned the same Household/Person IDs.

Appendix 1: Integerization algorithm

This appendix explains the algorithms and technical background behind the biased and minimally biased integerization approaches implemented in GO.

First, we introduce some notations used in the algorithm:

$\boldsymbol{w}^0$: initial weights (vector)

$\boldsymbol{w}^R$: calibrated real weights (vector)

$\boldsymbol{w}^I$: vector of integer part of $\boldsymbol{w}^R$(vector)

$\boldsymbol{w}^F$: fractional part of $\boldsymbol{w}^R$ (vector)

$\boldsymbol{w}^B$: binary weights (0 or 1) (vector)

$\boldsymbol{w}^C$: calibrated integer weights (vector)

$\boldsymbol{w}_i$: the weight for survey respondent i

$\boldsymbol{A}$: survey data recast to flag congruity with each benchmark via a series of dummy variables (0/1) (matrix)

$\boldsymbol{A}_{\cdot j}$: survey data for benchmark j for all survey respondents i (vector)

$\boldsymbol{B}$: the set of benchmarks values (vector)

$\boldsymbol{B}_j$: the value of benchmark j

B_T: benchmark total

$\boldsymbol{L}$: lower bound for each calibrated weight (constant vector)

$\boldsymbol{U}$: upper bound for each calibrated weight (constant vector)

TAE*: minimum achievable TAE given A, B , L, U and integer calibration weights

Notes:

- Matrix $\boldsymbol{A}$ has one row per survey respondent and one column per calibration benchmark.
- All vectors have one entry per survey respondent.
- With the exception of $\boldsymbol{w}^0$, $\boldsymbol{A}$, $\boldsymbol{L}$ and $\boldsymbol{U}$, all of the above values are area-specific.

The Linear Programming approach to integerization

Following Vovsha et al. (2014), Choupani and Mamdoohi (2016) and Espuny-Pujol et al (2018), GO obtains calibrated integer weights via the following five steps.

1. Compute the minimum achievable TAE of the integer solution for the benchmark and weight constraints, making no attempt to minimise the change from initial weights in the calibration solution:

$$\mathrm{TAE}^* = \min \ |\boldsymbol{w}^C\boldsymbol{A} - \boldsymbol{B}|$$

$$\text{subject to } \left\{ \begin{matrix} \sum \boldsymbol{w}^C\boldsymbol{A} = B_T \\ \boldsymbol{L} \leq \boldsymbol{w}^C \leq \boldsymbol{U};\ \text{integer } \boldsymbol{w}^C \end{matrix} \right\}$$

This problem is solved using linear programming methods.

Note that, if using integer benchmarks and integer boundary constraints, the minimum TAE of the integer and real calibration solutions will be identical.

2. Calibrate real weights $\boldsymbol{w}^R$, minimizing the change from initial weights, as measured using a chi-square loss function, finding a solution that satisfies the calibration constraints and matches the already identified minimum achievable TAE.

$$\min \sum_i (\boldsymbol{w}_i^R - \boldsymbol{w}_i^0)^2 \ / \boldsymbol{w}_i^0$$

$$\text{subject to } \sum(\boldsymbol{w}^R \boldsymbol{A} - \boldsymbol{B}) = \text{TAE}^*;\ \boldsymbol{L} \leq \boldsymbol{w}^R \leq \boldsymbol{U}$$

This problem is solved using *quadratic* linear programming methods.

The real weights, $\boldsymbol{w}^R$, provide a close approximation to the integer solution, but need to be rounded to an integer value to provide an integer solution. The next two stages decompose this problem to focus on rounding the fractional part, $\boldsymbol{w}^F$, of the already derived real weights $\boldsymbol{w}^R$.

3. Split the calibrated real weights $\boldsymbol{w}^R$ into integer and fractional parts so that:

$$\boldsymbol{w}_i^R = \boldsymbol{w}_i^I + \boldsymbol{w}_i^F$$

Using the integer part of the weights only will lead to a solution that no longer has minimum achievable TAE: $\sum_{ij}(\boldsymbol{A}_{ij}\, \boldsymbol{w}_i^I - \boldsymbol{B_j}) \neq \text{TAE}^*$. To achieve the minimum TAE the fractional part of the calibration weights, $\boldsymbol{w}^F$, must be appropriately rounded to values of 0 or 1. This is the focus of the next stage of the algorithm.

4. Search for a set of 0/1 binary weights, $\boldsymbol{w}^B$, that sum to the already identified minimum TAE, whilst minimizing their difference from the fractional part $\boldsymbol{w}^F$ of the calibration weights $\boldsymbol{w}^R$.

This is achieved by minimizing a loss function:

$$\min\big(f(\boldsymbol{w}^B, \boldsymbol{w}^F)\big)$$

$$\text{subject to } \ \sum\big(\boldsymbol{w}^B \boldsymbol{A} - \underline{\boldsymbol{B}}\big) = \text{TAE}^*;\ \boldsymbol{w}_i^B \in \{0,1\}$$

$$\text{where } \ \underline{\boldsymbol{B}} = \boldsymbol{B} - \boldsymbol{w}^I \boldsymbol{A}$$

The loss function $f(\boldsymbol{w}^B, \boldsymbol{w}^F)$ measures the distance between $\boldsymbol{w}^B$ and $\boldsymbol{w}^F$. $\underline{\boldsymbol{B}}$, the residual benchmarks, are the error in fit to the benchmarks left after multiplying the survey $\boldsymbol{A}$ by the integer part of $\boldsymbol{w}^R$.

This problem is solved using linear or quadratic programming methods depending upon the loss function chosen. The implementation and implications of alternative loss functions are discussed below.

5. Combine the integer-part of the original calibration weights, $\boldsymbol{w}^I$, with the now calibrated binary weights, $\boldsymbol{w}^B$, to derive the final calibrated integer weights, $\boldsymbol{w}^C$:

$$\boldsymbol{w}_i^C = \boldsymbol{w}_i^I + \boldsymbol{w}_i^B$$

Biased loss functions

Previous literature has proposed three alternative loss functions for use when calibrating $\boldsymbol{w}^B$, each of which produces a biased solution for the integerization problem outlined in step 4 above:

- $\min\left(-\sum_i \boldsymbol{w}_i^B \times \boldsymbol{w}_i^F\right)$ (Choupani and Mamdoohi, 2015)
- $\min\left(-\sum_i \boldsymbol{w}_i^B \times \ln\left(\boldsymbol{w}_i^F\right)\right)$ (Vovsha et al. 2014)
- $\min\left(\sum_i \left(\boldsymbol{w}_i^B - \boldsymbol{w}_i^F\right)^2 / \boldsymbol{w}_i^F\right)$ (Espuny-Pujol et al. 2018)

The first two loss functions minimize the linear distance between the binary weights $\boldsymbol{w}^B$ and fractional weights $\boldsymbol{w}^F$. The third minimizes the quadratic distance between $\boldsymbol{w}^B$ and $\boldsymbol{w}^F$.

All three loss functions are biased in the way that they handling the rounding of $\boldsymbol{w}^F$. There are two reasons for this, which are explored below.

a) Biasedness due to rounding threshold

Table 1 provides the basis for a simple illustrative example. It presents a small set of survey data to be calibrated, which contains the person-level variables Age and Sex. Table 1 also reports $\boldsymbol{w}^F$, the *fractional part* of the calibrated real weights derived at the end of stage 3 above.

Table 1 Calibrated survey data: example 1

Person (i)	Age	Sex	Fractional part of calibrated weight $\boldsymbol{w}^F$	Linear programming solution $\boldsymbol{w}^B$
1	Y	M	0.8	1
2	Y	F	0.2	0
3	O	M	0.2	0
4	O	F	0.8	1

Table 2 reports the residual benchmark targets left once account has been taken of the integer part of the calibrated real weights, $\underline{\boldsymbol{B}}$ (where $\underline{\boldsymbol{B}} = \boldsymbol{B} - \boldsymbol{w}^I\boldsymbol{A}$):

Table 2 Residual benchmark totals

	Benchmark			
	Age.Y	Age.O	Sex.M	Sex.F
$\underline{\boldsymbol{B}}$	1	1	1	1

For all three loss functions considered, linear programming will always generate the integer solution $\boldsymbol{w}^B$ shown in Table 1, because these are the integer weights closest to the fractions $\boldsymbol{w}_i^F$.

However, always rounding large fractions up to 1 and small fractions down to 0 is biased, because the expectation is:

$$E(\boldsymbol{w}^B) = \boldsymbol{w}^F \neq \boldsymbol{w}^B$$

In an unbiased solution the likelihood of rounding up would be proportional to the size of the fraction being rounded.

Bias due to rounding threshold becomes problematic when $\boldsymbol{w}^F$ does not vary randomly within and between population sub-groups, including sub-groups of interest defined by a combination of benchmark and non-benchmark survey variables. This tendency is most common when the number of respondents in the survey being calibrated greatly out-numbers the benchmark total, B_T, leading to the majority of real-valued calibration weights falling between 0 and 1.

Biased integerization causes the members of population sub-groups with the lowest real-valued calibration weights to be disproportionately rounded down, whilst those with the highest real-valued calibration weights will be disproportionately rounded up. As a result, biased rounding can lead to particular population sub-groups being over/under-represented in the final integer calibration solution. Even population sub-groups that map precisely onto calibration benchmarks can suffer from this problem, unless the final calibration solution fully satisfies the benchmark constraints.

b) Biasedness due to order of rounding

Each person (or household) in the survey falls within a population sub-group defined by a cell in the *n*-way tabulation of benchmark variables. For example, the benchmark variables in Table 2 give rise to the *n*-way tabulation of Age by Sex. Within this table is the cell 'Young Male'. All young male survey respondents will fall within this cell of the *n*-way benchmark table.

All persons (or households) from the survey that fall in the same cell of the *n*-way tabulation of benchmark variables, and also share the same initial weight, $\boldsymbol{w}^0$, will end up with the same real, and hence fractional, calibration weight.

For example, Table 3 contains the young male respondents from a survey that share the same age, sex and initial survey weights. As a result they all also have the same fractional calibration weight, $\boldsymbol{w}^F$, of 0.5.

However, the fact that, for example, certain survey respondents share the same age, sex and calibration weight does not mean that they share their other characteristics in common. As illustrated in Table 3, they might still differ in other ways, such as by health-status (well or unwell).

Table 3 Calibrated survey data: example 2

Person (i)	Age	Sex	Health	Initial survey weight ($\boldsymbol{w}^0$)	Fractional part of calibrated weight ($\boldsymbol{w}^F$)	Linear Programming solution ($\boldsymbol{w}^{B_1}$)	Alternative valid solution ($\boldsymbol{w}^{B_2}$)
1	Y	M	U	1.2	0.5	0	1
2	Y	M	W	1.2	0.5	1	0
3	Y	M	W	1.2	0.5	1	0
4	Y	M	U	1.2	0.5	0	1

With all three biased loss functions, linear programming will *always* generate the same fixed integer solution shown in Table 3, $\boldsymbol{w}^{B_1}$. Another integer solution exists that also minimizes distance whilst satisfying the constraints: $\boldsymbol{w}^{B_2}$. However, this latter solution will never be generated because, when faced with a choice regarding the rounding of a set of identical fractions, linear programming makes this choice based upon a deterministic ordering. Consequently the linear programming solution will be biased because $E(\boldsymbol{w}^B) = \boldsymbol{w}^F \neq \boldsymbol{w}^{B_1}$.

This is undesirable in so far as the bias in rounding correlates with non-benchmarked survey attributes of interest. This is illustrated in Table 3, where the first rounding solution, $\boldsymbol{w}^{B_1}$, leads to an estimate of 2 well persons and 0 unwell persons, whilst the second, $\boldsymbol{w}^{B_2}$, leads to an estimate of two unwell persons and 2 well persons. Using unbiased rounding the expectation is that after rounding there would be 1 well and 1 unwell person. Note that rounding is guaranteed unbiased for all persons in a given cell of the n-way benchmark table, even if their fractional weights differ (due to differing initial weights).

The arguments set out above also apply to the far less common scenario where two or more survey respondents with identical initial survey weights end up sharing identical calibration weights, despite being drawn from different cells of the *n*-way tabulation of benchmark variables. However, in this case competing calibration constraints might lead to the rounding being minimally biased rather than truly unbiased.

Minimally biased loss function

Summary

A stochastic procedure is adopted to achieve minimally biased rounding. In short, fractional weights are rounded up to one with a likelihood directly proportional to their fractional value. This means that small fractions will have a small chance of being rounded up and a large chance of being rounded down, whilst larger fractions will have a large chance of being rounded up and a small chance of being rounded down. In the case of tied calibration weights all values will have the same chance of being rounded up or down. Note that, as a result of the stochastic nature of this approach GO might select a different solution each time it is run, should multiple integer calibration solutions exist.

In simple cases (e.g. few benchmark variables; relatively loose lower and upper bounds on the weights permissible; large surveys), the stochastic rounding process implemented in GO will be unbiased: $E(\boldsymbol{w}^B) = \boldsymbol{w}^F$. Hence the probability of $\boldsymbol{w}_1^F = 0.8$ being rounded up would be four times larger than the probability of rounding up $\boldsymbol{w}_2^F = 0.2$.

For more complex cases, such as those involving multiple benchmark variables, tighter bounds on weights and small surveys, the need to simultaneously satisfy a series of competing constraints on the calibration solution is likely to limit the ability to fully implement unbiased rounding. Consequently, for more complex cases $w_1^F = 0.8$ *may not* have exactly an 80% chance of being rounded up to 1. However, in so far as the calibration constraints allow, the rounding will still be as unbiased as possible. Hence we describe the resulting solution as guaranteed 'minimally' biased rather than guaranteed unbiased.

Implementation details

The algorithm for minimally biased rounding splits Step 4 of the 'Linear Programming approach to integerization' already outlined above into five sub-steps. These sub-steps are illustrated in Table 4 and explained in the text that follows.

Table 4 algorithm steps

LP integerization Step	Value	Person 1	Person 2	Person 3	Person 4	Sum	Target
3	$\boldsymbol{w}_i^R$	1.8	3.2	2.2	0.8	8	8
3	$\boldsymbol{w}_i^I$	1	3	2	0	6	8
3	$\boldsymbol{w}_i^F$	0.8	0.2	0.2	0.8	2	2
4.1	$\boldsymbol{b}_i$	1	0	1	1	3	2
4.2, 4.3	$\boldsymbol{z}_i$	1.4	0.4	0.2	1.1	3.1	2
4.4	initial $\boldsymbol{w}_i^B$	1	1	1	1	4	2
4.5	calibrated $\boldsymbol{w}_i^B$	1	1	0	0	2	2
5	$\boldsymbol{w}_i^C$	2	4	2	0	8	8

Having rounded the original fractional weights to 0 or 1 in a way which satisfies the calibration constraints (steps 4.1 to 4.5), all that remains is to calculate the final integer calibration weight reported at the foot of the table by adding the rounded fractional weight (calibrated $\boldsymbol{w}_i^B$) to the integer part of the original real calibration weight (Step 5).

The real, integer and fractional calibration weights reported at the head of Table 4 are the output from Step 3 of the LP approach to integerization. The target total for the final integer calibration weights, $\boldsymbol{w}^C$, is the sum of the real calibration weights, $\boldsymbol{w}^R$ from Step 3.

In Step 4.1 each fractional weight, $\boldsymbol{w}^F$, is rounded up to 1 with a likelihood proportional to the fractional value, and if not rounded up, is rounded down to 0.

The sum of the rounded binary weights, $\boldsymbol{b}$, needs to equal the sum of the fractional weights, $\boldsymbol{w}^F$. Unfortunately, given the stochastic nature of the rounding process, the sum of the rounded 0/1 values at this stage is not guaranteed to match this target.

This is addressed by switching randomly selected values of $\boldsymbol{b}$ from 1 to 0 (and vice versa) until they sum to the required residual benchmark total (Steps 4.2 to 4.5), which itself is framed here as a calibration problem.

Problematically, conventional LP solutions, when required to choose which of a set of rounded values of 1 to switch to 0 (and vice versa), do so in a deterministic rather than random order, leading to the potential for biased rounding previously demonstrated. Therefore a random number is added to each rounded value of 1 and 0, in a way that ensures that all of the 1s, post adjustment, end up with a higher value than any of adjusted values of 0 (Steps 4.2 and 4.3). This provides the introduction of a random ordering to the LP calibration process, with the highest post adjustment values prioritised for rounding to 1 and the lowest values for rounding down to 0 (Step 4.5). As shown in Table 4, the set of binary weights, $\boldsymbol{w}^B$, that result from this process sum as required to the same total as the sum of the fractional weights, $\boldsymbol{w}^F$.

Having converted the fractional weights into binary weights with the same total, all that remains in Step 5 is to add the calibrated binary weights , $\boldsymbol{w}^B$, to the integer part of the original real calibration weights, $\boldsymbol{w}^I$. This yields the desired integer calibration weights, $\boldsymbol{w}^C$, which sum to the same total (8 in Table 4) as the original real calibration weights, $\boldsymbol{w}^R$.

More formally, the implemented solution can be described as follows:

LP Approach Step 3

Split the real-weighted calibration solution $\boldsymbol{w}^R$ into its constituent integer ($\boldsymbol{w}^I$) and fractional ($\boldsymbol{w}^F$) parts

LP Approach Step 4

Step 4.1 For each fractional weight, $\boldsymbol{w}_i^F$, independently sample a binary number $\boldsymbol{b}_i$ $(0 \text{ or } 1)$ with probability $\boldsymbol{w}_i^F$ of being 1.

Step 4.2 For each $\boldsymbol{b}_i = 0$, randomly sample a real-valued number from the uniform distribution $U(0,0.5)$, denoted by $\boldsymbol{z}_i$

Step 4.3 For each $\boldsymbol{b}_i = 1$, randomly sample a real-valued number from the uniform distribution $U(1.0,1.5)$, denoted by $\boldsymbol{z}_i$.

Note that $\boldsymbol{z}_i$ for $\boldsymbol{b}_i = 1$ is always larger than for $\boldsymbol{b}_i = 0$.

Step 4.4 Arbitrarily initialise $\boldsymbol{w}^B$ by assigning each weight a value of 1. (The initial value makes no difference to the final solution, provided it is a constant.)

Step 4.5 Calibrate each $\boldsymbol{w}_i^B$ to find a solution with the minimum achievable TAE determined in Stage 1 of the basic approach, subject to meeting the residual benchmark constraints $\underline{\boldsymbol{B}}$, using the following loss function to minimize the change of $\boldsymbol{z}$ values:

$$\min_{\boldsymbol{w}_i^B} \left(-\sum_i \boldsymbol{z}_i \times \boldsymbol{w}_i^B\right)$$

$$\text{subject to } \sum \left|\boldsymbol{w}^B \boldsymbol{A} - \underline{\boldsymbol{B}}\right| = \text{TAE}^*; \quad \boldsymbol{w}_i^B \epsilon \{0,1\}$$

LP Approach Stage 5

Compute the calibrated integer weights to be returned by GO using $\boldsymbol{w}_i^C = \boldsymbol{w}_i^I + \boldsymbol{w}_i^B$.

References

Choupani, A. & Mamdoohi, A. (2015). Population synthesis in activity-based models. *Transportation Research Record, 2493,* 1-10.

Espuny-Pujol, F., Morrissey, K., & Williamson, P. (2018). A global optimisation approach to range-restricted survey calibration, *Statistics and Computing,* 28, 427-439.

Vovsha, P., Hicks, J., Paul, B., Livshits, V., Maneva, P., & Jeon, K. (2014). New features of population synthesis, *Proceedings of the 94th Annual Meeting of the Transportation Research Board*, Transportation Research Board of the National Academies, Washington, D.C.

Appendix 2: R-code for generating demonstration datasets

```
## Create a randomly generated pseudo-survey for illustrative purposes
set.seed(10)
Person <- 1:1000                              # Generate person ID
Age <- sample(1:5, 1000, replace = T)         # Generate an Age variable
Sex <- sample(c(1,2), 1000, replace = T)      # Generate a Sex variable
survey_data <- data.frame(Person, Age, Sex) # Combine into a dataframe
head(survey_data)

## Create a randomly generated set of benchmark data (for 5 areas)

# (a) Generate area codes and populations
Area <- paste("Area", 1:5, sep = ".")     # Unique area code
Pop <- rep(300, 5)                        # Total population

# (b) Generate area-level benchmark targets for each Age category
Age.bm <- t( replicate(5, table(sample(1:5, 300, replace = T,
             prob = c(0.1, 0.2, 0.3, 0.2, 0.2) ) ) ) )

# (c) Generate area-level benchmark targets for each Sex category
Sex.bm <- t( replicate(5, table(sample(c(1,2), Pop, replace = T) ) ) )

# (d) Combine into a benchmark data frame with named columns
benchmark <- data.frame(Area, Pop,Age.bm, Sex.bm)
names(benchmark) <- c("Area", "Pop", paste("Age", 1:5, sep = "."),
                      paste("Sex", c(1,2), sep = "."))

# (e) Examine resulting benchmark data
head(benchmark)
```