The Stata Journal (*yyyy*) *vv*, Number *ii*, pp. 1–55

Updates to the ipfraking ecosystem

Stanislav Kolenikov
Abt Associates
stas_kolenikov@abtassoc.com

Abstract. Kolenikov (2014) introduced the package `ipfraking` for iterative proportional fitting (raking) weight calibration procedures for complex survey designs. This article briefly describes the original package and updates to the core program, and documents additional programs that are used to support the process of creating survey weights in author's production code.

Keywords: st0001, survey, calibration, weights, raking

1 Introduction and background

Large scale social, behavioral and health data are often collected via complex survey designs that may involve stratification, multiple stages of selection and/or unequal probabilities of selection (Korn and Graubard 1995, 1999). In an ideal setting, varying probabilities of selection are accounted for by using the Horvitz-Thompson estimator of the totals (Horvitz and Thompson 1952; Thompson 1997), and the remaining sampling fluctuations can be further ironed out by post-stratification (Holt and Smith 1979). However, on top of the planned differences in probabilities of obtaining a response from a sampled unit, non-response is a practical problem that has been growing more acute in recent years (Groves et al. 2001; Pew Research Center 2012). The analysis weights that are provided along with the public use microdata by data collecting agencies are designed to account for unequal probabilities of selection, non-response, and other factors affecting imbalance between the population and the sample, thus making the analyses conducted on such microdata generalizable to the target population.

Earlier work (Kolenikov 2014) introduced a Stata package called `ipfraking` that implements calibration of survey weights to known control totals to ensure that the resulting weighted data are representative of the population of interest. The process of calibration is aimed at aligning the sample totals of the key variables with those known for the population as a whole. The remainder of this section provides a condensed treatment of estimation with survey data using calibrated weights; a full description was provided in the previous paper.

For a given finite population $\mathcal{U}$ of units indexed $i = 1, \ldots, N$, the interests of survey statisticians often lie in estimating the population total of a variable Y:

$$T[Y] = \sum_{i \in \mathcal{U}} Y_i \tag{1}$$

A sample $\mathcal{S}$ of n units indexed by $j = 1, \ldots, n$ is taken from $\mathcal{U}$. If the probability to select the i-th unit is known to be π_i, then the *probability weights*, or *design weights*,

 st0001

are given by the inverse probability of selection:

$$w_{di} = \pi_i^{-1} \tag{2}$$

where subscript d stands for *design* probabilities of selection. With these weights, an unbiased (design-based, non-parametric) estimator of the total (1) is (Horvitz and Thompson 1952)

$$t[y] = \sum_{j \in \mathcal{S}} \frac{y_j}{\pi_j} \equiv \sum_{j \in \mathcal{S}} w_{dj} y_j \tag{3}$$

Probability weights protect the end user from potentially informative sampling designs, in which the probabilities of selection are correlated with outcomes, and relieve the user from the need to fully account for the sampling design variables in their analysis, as is required in methods such as multilevel regression with post-stratification (Park et al. 2004). Design-based methods generally ensure that inference can be generalized to the finite population even when the statistical models used by analysts and researchers are not specified correctly (Pfeffermann 1993; Binder and Roberts 2003).

Often, survey statisticians have auxiliary information on the units in the frame, and such information can be included at the sampling stage to create more efficient designs. Unequal probabilities of selection are then controlled with probability weights, implemented as `[pw=`*exp*`]` in Stata (and can be permanently affixed to the data set with `svyset` command).

In many situations, however, usable information is not available beforehand, and may only appear in the collected data. For example, the census totals of the age and gender distribution of the population may exist, but age and gender of the sampled units is unknown until the survey measurement is taken on them. It is still possible to capitalize on this additional data by adjusting the weights in such a way that the reweighted data conforms to these known figures. The procedures to perform these reweighting steps are generally known as *weight calibration* (Deville and Särndal 1992; Deville et al. 1993; Kott 2006, 2009; Särndal 2007).

Suppose there are several (categorical) variables, referred to as *control variables*, that are available for both the population and the sample (age groups, race, gender, educational attainment, etc.). Weight calibration aims at adjusting the weights via an iterative optimization so that the *control totals* for the control variables $\mathbf{x}_j = (x_{1j}, \dots, x_{pj})$, obtained with the calibrated weights w_{cj}, align with the known population totals:

$$\sum_{j \in \mathcal{S}} w_{cj} \mathbf{x}_j = T[\mathbf{X}] \tag{4}$$

The population totals of the control variables in the right hand side of (4) are assumed to be known from a census or a higher quality survey. Deville and Särndal (1992) framed the problem of finding a suitable set of weights as that of constrained optimization with the control equations (4) serving as constraints, and optimization targeted at making the discrepancy between the design weights w_{dj} and calibrated weights w_{cj} as close as possible, in a suitable sense.

The package `ipfraking` (Kolenikov 2014) implements a popular calibration algorithm, known as *iterative proportional fitting*, or *raking*, which consists of iterative updating (post-stratification) of each of the margins. (For an in-depth discussion of distinctions between raking and post-stratification, see Kolenikov (2016).) Since 2014, the continuing code development resulted in additional features that this update documents.

2 Updates to `ipfraking` program and package

Listed below is the full syntax of `ipfraking`, with the minimal description of options, and new features are discussed in a dedicated section. For additional details, please refer to Kolenikov (2014).

2.1 Syntax of `ipfraking`

`ipfraking` [*if*] [*in*] [*weight*] `,` `ctotal(`*matname* [*matname* ...]`)` [
`generate(`*newvarname*`)` `replace` `double` `iterate(#)` `tolerance(#)`
`ctrltolerance(#)` `trace` `nodivergence` `trimhiabs(#)` `trimhirel(#)`
`trimloabs(#)` `trimlorel(#)` `trimfrequency(once|sometimes|often)` `double`
`meta` `nograph`]

Note that the weight statement `[pw=`*varname*`]` is required, and must contain the initial weights.

Required options

`ctotal(`*matname* [*matname* ...]`)` supplies the names of the matrices that contain the control totals, as well as meta-data about the variables to be used in calibration.

❑ Technical note

The row and column names of the control total matrices (see [P] **matrix rownames**) should be formatted as follows.

- `rownames`: the name of the control variable
- `colnames`: the values of the control variable. The sets of values must be identical in the control total matrix and in the data set.
- `coleq`: the name of the variable for which the weighted total is computed. Typically it is identically equal to 1, and the examples below use conventionally named variable `_one`. See however Kolenikov and Hammer (2015) for other examples.

Detailed examples of the matrices created by manually entering the numbers, or using results of estimation commands with external, large scale survey data were given in Kolenikov (2014).

❑

`generate(`*newvarname*`)` contains the name of the new variable to contain the raked weights.

`replace` indicates that the weight variable supplied in the `[pw=`*varname*`]` expression should be overwritten with the new weights.

One and only one of `generate()` or `replace` must be specified.

Linear calibration

`linear` requests linear calibration of weights.

See section 6 for details, examples, and practical suggestions.

Options to control convergence

`tolerance(`#`)` defines convergence criteria (the change of weights from one iteration to next). The default is 10^{-6}.

`iterate(`#`)` specifies the maximum number of iterations. The default is 2000.

`nodivergence` overrides the check that the change in weights is greater at the current iteration than in the previous one, i.e., ignores this termination condition. It is generally recommended, especially in calibration with simultaneous trimming.

`ctrltolerance(`#`)` defines the criterion to assess the accuracy of the control totals. It does not impact iterations or convergence criteria, but rather only triggers alerts in the output. The default value is 10^{-6}.

`trace` requests a trace plot to be added.

Trimming options

By default, no weight trimming is applied. To reduce variability of weights and potential design effects, the following options can be specified.

`trimhiabs(`#`)` specifies the upper bound U on the greatest value of the raked weights. The weights that exceed this value will be trimmed down, so that $w_{3j} \le U$ for every $j \in \mathcal{S}$.

`trimhirel(`#`)` specifies the upper bound u on the adjustment factor over the baseline weight. The weights that exceed the baseline times this value will be trimmed down, so that $w_{3j} \le uw_{1j}$ for every $j \in \mathcal{S}$.

`trimloabs(`#`)` specifies the lower bound L on the smallest value of the raked weights.

The weights that are smaller than this value will be increased, so that $w_{3j} \geq L$ for every $j \in \mathcal{S}$.

`trimlorel(#)` specifies the lower bound l on the adjustment factor over the baseline weight. The weights that are smaller than the baseline times this value will be increased, so that $w_{3j} \geq l w_{1j}$ for every $j \in \mathcal{S}$.

`trimfreqency(`*keyword*`)` specifies when the trimming operations are to be performed. The following keywords are recognized:

`often` means that trimming will be performed after each marginal adjustment.

`sometimes` means that trimming will be performed after a full set of variables has been used for post-stratification. This is the default behavior if any of the numeric trimming options above are specified.

`once` means that trimming will be performed after the raking process is declared to have converged.

The numeric trimming options `trimhiabs(#)`, `trimhirel(#)`, `trimloabs(#)`, `trimlorel(#)` can be specified in any combination, or entirely omitted to produce untrimmed weights.

Miscellaneous options

`double` specifies that the new variable named in `generate()` option should be generated as double type. See [D] **data types**.

`meta` puts information taken by `ipfraking` as inputs and produced throughout the process into characteristics stored with the variable specified in `generate()` option.

`nograph` omits the histogram of the calibrated weights, which can be used to speed up `ipfraking` (e.g., in replicate weight production).

2.2 New features of `ipfraking`

Reporting of results and errors by `ipfraking` was improved in several directions.

1. The discrepancy for the worst fitting category is now being reported.

2. The number of trimmed observations is reported.

3. If `ipfraking` determines that the categories do not match in the control totals received from `ctotals()` and those found in the data, a full listing of categories is provided, and the categories not found in one or the other are explicitly shown.

Linear calibration (Case 1 of Deville and Särndal (1992)) is provided with `linear` option. The weights are calculated analytically:

$$w_{j,\mathrm{lin}} = w_{dj}(1 + \mathbf{x}_j'\lambda), \quad \lambda = \Big(\sum_{j\in\mathcal{S}} w_{dj}\mathbf{x}_j\mathbf{x}_j'\Big)^{-1}(T[\mathbf{X}] - t[X]) \tag{5}$$

Since no iterative optimization is required, linear calibration works very fast. However it has an undesirable artefact of potentially producing negative weights, as the range of weights is not controlled. (As raking works by multiplying the currents weights by positive factors, if the input weights are all positive, the output weights will be positive as well.) Negative weights are not allowed by the official `svy` commands or commands that work with `[pweights]`. In many tasks, running linear weights first, pulling up the negative and small positive weights (`replace weight = 1 if weight <= 1`) and re-raking using the "proper" iterative proportional fitting runs faster than raking from scratch. An example of linearly calibrated weights is given below in Section 6.

Option `meta` saves more information in characteristics of the calibrated weight variables. Using Example 3 from Kolenikov (2014) with trimming options, we have:

```
. capture drop rakedwgt3

. ipfraking [pw=finalwgt], gen( rakedwgt3 ) ///
>     ctotal( ACS2011_sex_age Census2011_region Census2011_race ) ///
>     trimhiabs(200000) trimloabs(2000) meta
 Iteration 1, max rel difference of raked weights = 14.95826
 Iteration 2, max rel difference of raked weights = .21474256
 Iteration 3, max rel difference of raked weights = .02754514
 Iteration 4, max rel difference of raked weights = .00511347
 Iteration 5, max rel difference of raked weights = .00095888
 Iteration 6, max rel difference of raked weights = .00018036
 Iteration 7, max rel difference of raked weights = .00003391
 Iteration 8, max rel difference of raked weights = 6.377e-06
 Iteration 9, max rel difference of raked weights = 1.199e-06
 Iteration 10, max rel difference of raked weights = 2.254e-07
The worst relative discrepancy of  3.0e-08 is observed for race == 3
Target value =   20053682; achieved value =   20053682
Trimmed due to the upper absolute limit: 5 weights.

   Summary of the weight changes

              |    Mean    Std. dev.    Min        Max       CV
--------------+----------------------------------------------------
Orig weights  |    11318       7304      2000       79634   .6453
Raked weights |    22055      18908      4033      200000   .8573
Adjust factor |   2.1486               0.9220     18.9828

. char li rakedwgt3[]
  rakedwgt3[source]:          finalwgt
  rakedwgt3[objfcn]:          2.25435521346e-07
  rakedwgt3[maxctrl]:         3.00266822363e-08
  rakedwgt3[converged]:       1
  rakedwgt3[worstcat]:        3
  rakedwgt3[worstvar]:        race
  rakedwgt3[command]:         [pw=finalwgt], gen( rakedwgt3 ) ctotal( ACS2011_sex_age Census2011_region ..
  rakedwgt3[trimloabs]:       trimloabs(2000)
  rakedwgt3[trimhiabs]:       trimhiabs(200000)
  rakedwgt3[trimfrequency]:   sometimes
  rakedwgt3[hash1]:           2347674164
  rakedwgt3[mat3]:            Census2011_race
```

```
rakedwgt3[over3]:           race
rakedwgt3[totalof3]:        _one
rakedwgt3[Census2011_race]: 7.48567503861e-09
rakedwgt3[mat2]:            Census2011_region
rakedwgt3[over2]:           region
rakedwgt3[totalof2]:        _one
rakedwgt3[Census2011_region]:
                            3.00266822363e-08
rakedwgt3[mat1]:            ACS2011_sex_age
rakedwgt3[over1]:           sex_age
rakedwgt3[totalof1]:        _one
rakedwgt3[ACS2011_sex_age]: 4.13778410340e-09
rakedwgt3[note1]:           Raking controls used: ACS2011_sex_age Census2011_region Census2011_race
rakedwgt3[note0]:           1
```

The following characteristics are stored with the newly created weight variable (see [P] **char**).

`command`	The full command as typed by the user
matrix name	The relative matrix difference from the corresponding control total, see [D] **functions**
`trimhiabs`, `trimloabs`, `trimhirel`, `trimlorel`, `trimfrequency`	Corresponding trimming options, if specified
`maxctrl`	the greatest `mreldif` between the targets and the achieved weighted totals
`objfcn`	the value of the relative weight change at exit
`converged`	whether `ipfraking` exited due to convergence (1) vs. due to an increase in the objective function or reaching the limit on the number of iterations (0)
`source`	weight variable specified as the `[pw=]` input
`worstvar`	the variable in which the greatest discrepancy between the targets and the achieved weighted totals (`maxctrl`) was observed
`worstcat`	the category of the `worstvar` variable in which the greatest discrepancy was observed

For the control total matrices $\#= 1, 2, \ldots$, the following meta-information is stored.

`mat`#	the name of the control total matrix
`totalof`#	the multiplier variable (matrix `coleq`)
`over`#	the margin associated with the matrix (i.e., the categories represented by the columns)

Also, `ipfraking` stores the notes regarding the control matrices used, and which of the margins did not match the control totals, if any. See [D] **notes**.

2.3 Additional examples

Kolenikov and Hammer (2015) demonstrate how to solve the problem of creating weights that are constant within a larger unit (e.g., in a household survey, it may be desirable that all individuals within a household have the same weight). They achieve this as follows:

1. Control totals are defined at the individual level.

2. At the household level, the weighting multipliers are defined for each *individual* level control category as the number of individuals in a household that have the requisite demographic characteristics. Within the syntax of `ipfraking`, these are the variables identified by the `coleq` names of the control total matrices, and they vary from household to household.

3. Raking is performed at the household level by using `if household_head==1` subsetting of `ipfraking`.

4. The resulting weights are assigned to every individual in the household.

2.4 Utility programs

The original package `ipfraking` provided additional utility programs: `mat2do`, `xls2row` and `totalmatrices`.

Design effects

A new utility program `whatsdeff` was added to compute the unequal weighting design effects and margins of error, common tasks associated with describing survey weights. Specifically, the Transparency Initiative of the American Association for Public Opinion Research (AAPOR 2014) requires that

> For probability samples, the estimates of sampling error will be reported, and the discussion will state whether or not the reported margins of sampling error or statistical analyses have been adjusted for the design effect due to weighting, clustering, or other factors.

`whatsdeff` *weight_variable* [*if*] [*in*] , [`by(`*varlist*`)`]

The utility program `whatsdeff` calculates the apparent design effect due to unequal weighting,

$$\mathrm{DEFF}_{\mathrm{UWE}} = 1 + CV_w^2 = \texttt{1 + r(Var)/(r(mean))\^{}2}$$

using the returned values from `summarize` *weight_variable* (see `help return`). Additionally, it reports the effective sample size, $n/\mathrm{DEFF}_{\mathrm{UWE}}$, and also returns the margins of error for the sample proportions that estimate the population proportions of 10% and 50%.

```
. webuse nhanes2, clear
. whatsdeff finalwgt
```

Group	Min	Mean	Max	CV	DEFF	N	N eff
Overall	2000.00	11318.47	79634.00	0.6453	1.4164	10351	7307.97

```
. return list
scalars:
                  r(N) =  10351
              r(MOE10) =  .0068792766212984
              r(MOE50) =  .0114654610354974
       r(Neff_Overall) =  7307.97435325364
       r(DEFF_Overall) =  1.416397964696134
```

Design effects can also be broken down by a categorical variable:

```
. whatsdeff finalwgt, by(sex)
```

Group	Min	Mean	Max	CV	DEFF	N	N eff
sex							
Male	2000.00	11426.14	79634.00	0.6578	1.4326	4915	3430.94
Female	2130.00	11221.12	61534.00	0.6333	1.4010	5436	3880.01
Overall	2000.00	11318.47	79634.00	0.6453	1.4164	10351	7307.97

```
. return list
scalars:
                  r(N) =  10351
              r(MOE10) =  .0068792766212984
              r(MOE50) =  .0114654610354974
       r(Neff_Overall) =  7307.97435325364
       r(DEFF_Overall) =  1.416397964696134
        r(Neff_Female) =  3880.00710397866
        r(DEFF_Female) =  1.40102836266093
          r(Neff_Male) =  3430.938195872213
          r(DEFF_Male) =  1.432552765279559
```

The estimates of unequal weighting design effects that `whatsdeff` produces should be considered as a typical magnitude of a design effect. As pointed out by a referee, in many situations when survey variables are correlated with weights or with the variables that weight calibration is based upon, the actual design effects reported by postestimation command `estat effect` should be expected to be lower, provided that variance estimation methods take calibration into account properly (e.g., via replicate variance estimation, as described in Kolenikov (2010), or via `svy, vce(calibrate)` functionality of the official Stata `svy` suite available in Stata 15.1+). In other words, for most situations these estimates could be considered an upper bound on this design effect, since this calculation assumes that the weights are independent of the survey variable of descriptive interest. Situations can be envisioned, however, when weighting is counterproductive for efficiency of certain statistics, e.g., when a group that is oversampled by the sampling design requirements turns out to have the outcome variance smaller than that of the rest population; or when screening is applied within rare population sampling designs (Kalton and Anderson 1986; Kalton 2009).

Conversion of the matrices

One of the utility programs available in Kolenikov (2014), `mat2do`, was updated to provide the `notimestamp` option to omit the time stamps (which tend to unnecessarily throw off the project building and revision control systems).

A new program `totalmatrices` converts the control totals matrices between the formats expected by `ipfraking` and `svycal`.

`totalmatrices` *matrix_list* `,` [`svycal` `ipfraking` `stub(`*name*`)` `replace`]

`svycal` checks that the supplied matrix or matrices are compatible with `svycal` specification of totals as a matrix.

`ipfraking` checks that the supplied matrix or matrices are compatible with `ipfraking`.

`stub(`*name*`)` provides the naming convention for the converted control total matrices. If the conversion is from `ipfraking` to `svycal`, one matrix whose name is supplied in the `stub(...)` option will be created. If the conversion is from `svycal` to `ipfraking`, matrices corresponding to each variable will be created and have their names set to concatenation of the stub and the variable name.

To convert several matrices from `ipfraking` format to a single matrix in `svycal` format:

```
. totalmatrices ACS2011_sex_age Census2011_region Census2011_race, ///
>         ipfraking stub(alltotals) replace convert
It appears that the matrix ACS2011_sex_age is of ipfraking format.
It appears that the matrix Census2011_region is of ipfraking format.
It appears that the matrix Census2011_race is of ipfraking format.
You can now matrix list alltotals to check and then call svycal as:
   svycal [regress|rake] 11.sex_age 12.sex_age 13.sex_age 21.sex_age 2
> 2.sex_age 23.sex_age 1.region 2.region 3.region 4.region 1.race 2.ra
> ce 3.race [pw=finalwgt], generate(...) totals(alltotals) nocons
I suspect the following would be simpler and could work, too:
   svycal [regress|rake] ibn.sex_age ibn.region ibn.race [pw=finalwgt]
> , generate(...) totals(alltotals) nocons
```

To convert a single matrix compatible with `svycal` requirements for its `totals(`*matname*`)` format into a list of matrices compatible with `ipfraking`:

```
. totalmatrices alltotals, ipfraking stub(totmat_) replace convert
It appears that the matrix alltotals is of the svycal format.
Matrices created:
matrix list totmat_sex_age
matrix list totmat_region
matrix list totmat_race

. matrix list totmat_region

totmat_region[1,4]
            _one:     _one:     _one:     _one:
               1         2         3         4
region  40679030  49205289  85024007  53385843
```

Note that at the moment, `totalmatrices` does not handle conversion of interactions, which is arguably one of the greatest strengths of `svycal`. As noted in section 6.1, for interactions to work out with `ipfraking`, standalone variables need to be created, and `totalmatrices` would rather have the user do that.

2.5 New programs in the package

Two new programs are added to the package: `ipfraking_report` and `wgtcellcollapse`, and are documented in the subsequent sections of this article. The former provides reports on the raked weights, including summaries of the data with no weights, with the input weights, and with the calibrated weights. The latter creates a mostly automated flow of collapsing weighting cells that are too detailed (and hence have low sample sizes).

3 Excel reports on raked weights: `ipfraking_report`

`ipfraking_report using` *filename* `, raked_weight(`*varname*`)` [
`matrices(`*namelist*`) by(`*varlist*`) xls replace force`]

The utility command `ipfraking_report` produces a detailed report describing the raked weights, and places it into *filename*`.dta` file (or, if `xls` option is specified, both *filename*`.dta` and *filename*`.xls` files).

Along the way, `ipfraking_report` runs a regression of the log raking ratio w_{3j}/w_{1j} on the calibration variables. This regression is expected to have R^2 equal to or very close to 1, and residual variance equal to or very close to zero. This naturally produces obscenely high t-test values, but the purpose of this regression is not in establishing "significance" of any variable in explaining the outcome (which we know to be predicted with near certainty). Instead, the regression coefficients provide insights regarding which categories received greater vs. smaller adjustments (which in turn indicate lower response or coverage rates for the corresponding population subgroups). Conversely, control variables that are associated with relatively similar adjustment factors may be contributing relatively little to the weight adjustment, and may be candidates for removal from the list of control totals.

Using the working example from Kolenikov (2014), the example regression output is:

```
. ipfraking_report using rakedwgt3-report, raked_weight(rakedwgt3) replace by(_one)
Margin variable sex_age (total variable: _one; categories: 11 12 13 21 22 23).
Margin variable region (total variable: _one; categories: 1 2 3 4).
Margin variable race (total variable: _one; categories: 1 2 3).
Auxiliary variable _one (categories: 1).
file rakedwgt3-report.dta saved
      Source |       SS           df       MS      Number of obs   =    10,351
-------------+----------------------------------   F(10, 10340)    >  99999.00
       Model |  2086.13859        10  208.613859   Prob > F        =    0.0000
```

```
    Residual |  .78315703    10,340  .000075741   R-squared       =    0.9996
-------------+----------------------------------   Adj R-squared   =    0.9996
       Total |  2086.92175    10,350  .201634952   Root MSE        =     .0087

------------------------------------------------------------------------------
    __000003 |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
     sex_age |
         11  |   .0644365   .0002775   232.21   0.000     .0638925    .0649804
         12  |   .4545577   .0003154  1441.25   0.000     .4539395     .455176
         13  |   .6782466   .0002804  2418.71   0.000     .6776969    .6787963
         22  |   .3966406   .0003049  1300.84   0.000     .3960429    .3972383
         23  |   .7304392   .0002726  2679.97   0.000     .7299049    .7309734
             |
      region |
         NE  |  -.4455127   .0002536 -1756.49   0.000    -.4460099   -.4450155
         MW  |  -.4428144   .0002335 -1896.53   0.000    -.4432721   -.4423567
          W  |  -.6672675   .0002407 -2772.21   0.000    -.6677393   -.6667957
             |
        race |
      Black  |   .3360321   .0002848  1180.08   0.000     .3354739    .3365902
      Other  |   1.613276   .0006303  2559.34   0.000     1.612041    1.614512
             |
       _cons |   .5864801   .0002455  2388.48   0.000     .5859988    .5869614
------------------------------------------------------------------------------

Raking adjustments for sex_age variable:
  the smallest was        1.798 for category 21 (21)
  the greatest was        3.732 for category 23 (23)
Raking adjustments for region variable (1=NE, 2=MW, 3=S, 4=W):
  the smallest was        0.922 for category 4 (W)
  the greatest was        1.798 for category 3 (S)
Raking adjustments for race variable (1=white, 2=black, 3=other):
  the smallest was        1.798 for category 1 (White)
  the greatest was        9.023 for category 3 (Other)
```

It looks like `ipfraking` had to make greater adjustments to the weights of older females (`sex_age==23`, i.e., `sex==2 & age==3`; the adjustment factor for this category was 3.732 vs. the low of 1.798 for young women), and especially other race individuals (the adjustment factor was 9.023, vs. 1.798 for the whites). The diagnostic value is in the differences in the adjustment factors with the same variable; since no attempt is being made to average across the population or the sample or to assign the "base" variable, the absolute reported values of the adjustment factors may not be meaningful. In the example above, 1.798 figures both as the greatest adjustment factor of the region variable, and as the lowest adjustment factor for the race the and sex-by-age interaction. As is easily seen from regression output, this value is the exponent of the intercept `1.798=exp(0.586)`. Since all of the "estimates" of the region specific coefficients are negative, the lowest reported value is less than this baseline value. Since all of the "estimates" of the race and sex-by-age indicators are positive, all the category-specific adjustment factors are greater than this baseline value. This is an interplay of the base categories, and the differences in the demographic composition within each category of a control total variable vis-a-vis other weighting variables.

3.1 Options of ipfraking_report

raked_weight(*varname*) specifies the name of the raked weight variable to create the report for. This is a required option.

matrices(*namelist*) specifies a list of matrices (formatted as the matrices supplied to ctotal() option of ipfraking) to produce weighting reports for. In particular, the variables and their categories are picked up from these matrices; and the control totals/proportions are compared to those defined by the weight being reported on.

by(*varlist*) specifies a list of additional variables for which the weights are to be tabulated in the raking weights report. The difference with the matrices() option is that the control totals for these variables may not be known (or may not be relevant). In particular, by(_one), where _one is identically one, will produce the overall report.

xls requests exporting the report to an Excel file.

replace specifies that the files produced by ipfraking_report (i.e., the .dta and the .xls file if xls option is specified) should be overwritten.

force requires that a variable that may be found repeatedly (between the calibration variables supplied originally to ipfraking, the variables found in the independent total matrices(), and the variables without the control totals provided in by() option) is processed every time it is encountered. (Otherwise, it is only processed once.)

3.2 Variables in the raking report

The raking report file contains the following variables.

(*Continued on next page*)

Variable name	Definition
`Weight_Variable`	The name of the weight variable, `generate()`
`C_Total_Margin_Variable_Name`	The name of the control margin, `rowname` of the corresponding `ctotal()` matrix
`C_Total_Margin_Variable_Label`	The label of the control margin variable
`Variable_Class`	The role of the variable in the report: Raking margin: a variable used as a calibration margin (picked up automatically from the `ctotal()` matrix, provided `meta` option was specified) Other known target: supplied with `matrices()` option of `ipfraking_report` Auxiliary variable: additional variable supplied with `by()` option of `ipfraking_report`
`C_Total_Arg_Variable_Name`	The name of the multiplier variable
`C_Total_Arg_Variable_Label`	The label of the multiplier variable
`C_Total_Margin_Category_Number`	Numeric value of the control total category
`C_Total_Margin_Category_Label`	Label of the control total category
`C_Total_Margin_Category_Cell`	An indicator whether a weighting cell was produced by collapsing categories using `wgtcellcollapse`
`Category_Total_Target`	The control total to be calibrated to (the specific entry in the `ctotal()` matrix)
`Category_Total_Prop`	Control total proportion (the ratio of the specific entry in the `ctotal()` matrix to the matrix total)
`Unweighted_Count`	Number of sample observations in the category
`Unweighted_Prop`	Unweighted proportion
`Unweighted_Prop_Discrep`	Difference `Unweighted_Prop - Category_Total_Prop`
`Category_Total_SRCWGT`	Weighted category total, with input weight
`Category_Prop_SRCWGT`	Weighted category proportion, with input weight
`Category_Total_Discrep_SRCWGT`	Difference `Category_Total_SRCWGT -` `- Category_Total_Target`
`Category_Prop_Discrep_SRCWGT`	Difference `Category_Prop_SRCWGT -` `- Category_Total_Prop`
`Category_RelDiff_SRCWGT`	`reldif(Category_Total_SRCWGT, Category_Total_Target)`
`Overall_Total_SRCWGT`	Sum of source weights
`Source`	The name of the matrix from which the totals were obtained
`Comment`	Placeholder for comments, to be entered during manual review

For each of the input weights (`SRCWGT` suffix), raked weights (`RKDWGT` suffix) and raking ratio (the ratio of raked and input weights, `RKDRATIO` suffix), the following summaries are provided.

Variable name	Definition
Min_*WEIGHT*	Min of the respective weight
P25_*WEIGHT*	25th percentile of the respective weights
P50_*WEIGHT*	Median of the respective weights
P75_*WEIGHT*	75th percentile of the respective weights
Max_*WEIGHT*	Max of the respective weights
Mean_*WEIGHT*	Mean of the respective weights
SD_*WEIGHT*	Standard deviation of the respective weights
DEFF_*WEIGHT*	Apparent UWE DEFF of the respective weights

3.3 Example

Continuing with the example of calibration by region, race, and sex-by-age interaction, a glimpse of the raking report looks as follows.

```
. use rakedwgt3-report, clear
(Weighting report on rakedwgt3)
. list C_Total_Margin_Variable_Name C_Total_Margin_Category_Label ///
>        Category_Total_Target Category_Total_RKDWGT DEFF_SRCWGT DEFF_RKDWGT , ///
>        sepby( C_Total_Margin_Variable_Name )

     +-----------------------------------------------------------------------------+
     | C_Tota..   ~y_Label   Categor~t   Categor..   DEFF_SR~T   DEFF_RK~T |
     |-----------------------------------------------------------------------------|
  1. |  sex_age         11    41995394    41995394   1.2148059   1.6259899 |
  2. |  sex_age         12    42148662    42148662   1.2462168   1.5716613 |
  3. |  sex_age         13    26515340    26515340   1.2241095   1.5460785 |
  4. |  sex_age         21    41164255    41164255   1.2325105   1.5639529 |
  5. |  sex_age         22    43697440    43697440   1.1937826   1.5175312 |
  6. |  sex_age         23    32773080    32773080    1.233902    1.664307 |
     |-----------------------------------------------------------------------------|
  7. |   region         NE    40679030    40679030   1.3056639   1.3657837 |
  8. |   region         MW    49205289    49205289   1.3475551   1.4909581 |
  9. |   region          S    85024007    85024006   1.4950056   1.4912995 |
 10. |   region          W    53385843    53385844    1.459859   2.3772667 |
     |-----------------------------------------------------------------------------|
 11. |     race      White   1.784e+08   1.784e+08   1.4059259   1.4337901 |
 12. |     race      Black    29856865    29856865   1.5173846   1.5092533 |
 13. |     race      Other    20053682    20053682   1.3179136   1.2264706 |
     |-----------------------------------------------------------------------------|
 14. |     _one          1           .   2.283e+08   1.4164382   1.7349278 |
     +-----------------------------------------------------------------------------+
```

The last line, corresponding to the auxiliary variable `_one` identically equal to 1 (this variable was present in the data set as it was used by `ipfraking` as a multiplier), contains summaries for the sample as a whole. It is always recommended to include it (note the use of `ipfraking_report, ... by(_one)` option in the syntax in the previous section.)

Functionality of `ipfraking_report` is aimed at manual quality control review, which typically involves (i) categories with raking factors that differ the most (in the output), and (ii) the resulting report file in Excel, although for some aspects of automated quality control, it can be useful, as well.

4 Collapsing weighting cells: `wgtcellcollapse`

An additional new component of `ipfraking` package is a tool to semi-automatically collapse weighting cells, in order to achieve a required minimal size of the weighting cell. (A typical recommendation is to have cells of size 30 to 50.)

`wgtcellcollapse` *task* [*if*] [*in*] , [`task_options`]

where *task* is one of:

`define` to define collapsing rules explicitly

`sequence` to create collapsing rules for a sequence of categories

`report` to list the currently defined collapsing rules

`candidate` to find rules applicable to a given category

`collapse` to perform cell collapsing

`label` to label collapsed cells using the original labels after `wgtcellcollapse collapse`

4.1 Syntax of `wgtcellcollapse report`

`wgtcellcollapse report , variables(`*varlist*`)` [`break`]

`variables(`*varlist*`)` is the list of variables for which the collapsing rules are to be reported

`break` requires `wgtcellcollapse report` to exit with error when technical inconsistencies are encountered

4.2 Syntax of `wgtcellcollapse define`

`wgtcellcollapse define , variables(`*varlist*`)` [`from(`*numlist*`) to(`#`)` `label(`*string*`) max(`#`) clear`]

`variables(`*varlist*`)` is the list of variables for which the collapsing rule can be used

`from(`*numlist*`)` is the list of categories that can be collapsed according to this rule

`to(`#`)` is the numeric value of the new, collapsed category

`label(`*string*`)` is the value label to be attached to the new, collapsed category

`max(`#`)` overrides the automatically determined max value of the collapsed variable

`clear` clears all the rules currently defined

Let us demonstrate the two subcommands introduced so far with the following toy example.

```
. clear
.
. set obs 4
number of observations (_N) was 0, now 4
.
. gen byte x = _n
.
. label define x_lbl 1 "One" 2 "Two" 3 "Three" 4 "Four"
.
. label values x x_lbl
.
. wgtcellcollapse define, var(x) from(1 2 3) to(123)
.
. wgtcellcollapse report, var(x)
Rule (1): collapse together
  x == 1 (One)
  x == 2 (Two)
  x == 3 (Three)
  into x == 123 (123)
  WARNING: unlabeled value x == 123
```

For automated quality control purposes, the `break` option of `wgtcellcollapse report` can be used to abort the execution when technical deficiencies in the rules or in the data are encountered. In the above example, the label of the new category 123 was not defined. Should the `break` option be specified, this would be considered a serious enough deficiency to stop with an error:

```
. wgtcellcollapse report, var(x) break
Rule (1): collapse together
  x == 1 (One)
  x == 2 (Two)
  x == 3 (Three)
  into x == 123 (123)
  ERROR: unlabeled value x == 123
assertion is false
r(9);
.
. wgtcellcollapse define, var(x) clear
.
. wgtcellcollapse define, var(x) from(1 2 3) to(123) label("One through three")
.
. wgtcellcollapse report, var(x) break
Rule (1): collapse together
  x == 1 (One)
  x == 2 (Two)
  x == 3 (Three)
  into x == 123 (One through three)
```

4.3 Syntax of `wgtcellcollapse sequence`

`wgtcellcollapse sequence , variables(`*varlist*`) from(`*numlist*`) depth(#)`

`variables(`*varlist*`)` is the list of variables for which the collapsing rule can be used

`from(`*numlist*`)` is the sequence of values from which the plausible subsequences can be constructed

`depth(#)` is the maximum number of the original categories that can be collapsed

Continuing with the toy example introduced above, here is an example of moderate length sequences to collapse categories:

```
. clear
. set obs 4
number of observations (_N) was 0, now 4
. gen byte x = _n
. label define x_lbl 1 "One" 2 "Two" 3 "Three" 4 "Four"
. label values x x_lbl
. wgtcellcollapse sequence, var(x) from(1 2 3 4) depth(3)
. wgtcellcollapse report, var(x)
Rule (1): collapse together
  x == 1 (One)
  x == 2 (Two)
  into x == 212 (One to Two)
Rule (2): collapse together
  x == 2 (Two)
  x == 3 (Three)
  into x == 223 (Two to Three)
Rule (3): collapse together
  x == 3 (Three)
  x == 4 (Four)
  into x == 234 (Three to Four)
Rule (4): collapse together
  x == 1 (One)
  x == 2 (Two)
  x == 3 (Three)
  into x == 313 (One to Three)
Rule (5): collapse together
  x == 1 (One)
  x == 223 (Two to Three)
  into x == 313 (One to Three)
Rule (6): collapse together
  x == 3 (Three)
  x == 212 (One to Two)
  into x == 313 (One to Three)
```

(*Continued on next page*)

```
Rule (7): collapse together
  x == 2 (Two)
  x == 3 (Three)
  x == 4 (Four)
  into x == 324 (Two to Four)
Rule (8): collapse together
  x == 2 (Two)
  x == 234 (Three to Four)
  into x == 324 (Two to Four)
Rule (9): collapse together
  x == 4 (Four)
  x == 223 (Two to Three)
  into x == 324 (Two to Four)
```

Note how `wgtcellcollapse sequence` automatically created labels for the collapsed cells.

When creating sequential collapses, `wgtcellcollapse sequence` uses the following conventions in creating the new categories:

- First comes the length of the collapsed subsequence (up to `depth(#)`).
- Then comes the starting value of the category in the subsequence (padded by zeroes as needed).
- Then comes the ending value of the category in the subsequence (padded by zeroes as needed).

In the example above, rules 7 through 9 lead to collapsing into the new category 324. This should be interpreted as "the subsequence of length 3 that starts with category 2 and ends with category 4". A numeric value of the collapsed category that reads like 50412 means "the subsequence of length 5 that starts with category 4 and ends with category 12". In that second example, `wgtcellcollapse sequence` padded the value of 4 with an additional zero, so that the length of resulting collapsed category value is always (of digits of the sequence length) + twice (of digits of the greatest source category).

Note that `wgtcellcollapse sequence` respects the order in which the categories are supplied in the `from()` option, and does not sort them. If the categories are supplied in the order 2, 4, 1 and 3, then `wgtcellcollapse sequence` would collapse 2 with 4, 4 with 1, and 1 with 3:

```
. wgtcellcollapse define, var(x) clear
. wgtcellcollapse sequence, var(x) from(2 4 1 3) depth(2)
. wgtcellcollapse report, var(x)
Rule (1): collapse together
  x == 2 (Two)
  x == 4 (Four)
  into x == 224 (Two to Four)
```

```
Rule (2): collapse together
  x == 4 (Four)
  x == 1 (One)
  into x == 241 (Four to One)
Rule (3): collapse together
  x == 1 (One)
  x == 3 (Three)
  into x == 213 (One to Three)
```

4.4 Syntax of `wgtcellcollapse candidate`

`wgtcellcollapse candidate , variable(`*varname*`) category(`#`)` [max #]

`variable(`*varname*`)` is the variable whose collapsing rules are to be searched

`category(`#`)` is the category for which the candidate rules are to be identified

`max(`#`)` is the maximum value of the categories in the candidate rules to be returned

The rules found are quietly returned through the mechanism of `sreturn`, see [P] **return**, as they are intended to stay in memory sufficiently long for `wgtcellcollapse collapse` to evaluate each rule. Going back to the example from the previous section with sequential collapses of depth 3, we can identify the following candidates for categories 2, 212 (collapsed values of 1 and 2), and a non-existent category of 55:

```
. wgtcellcollapse candidate, var(x) cat(2)
. sreturn list
macros:
           s(goodrule) : "1 2 4 7 8"
              s(rule8) : "2:234=324"
              s(rule7) : "2:3:4=324"
              s(rule4) : "1:2:3=313"
              s(rule2) : "2:3=223"
              s(rule1) : "1:2=212"
                s(cat) : "2"
                  s(x) : "x"
. wgtcellcollapse candidate, var(x) cat(2) max(9)
. sreturn list
macros:
           s(goodrule) : "1 2 4 7"
              s(rule7) : "2:3:4=324"
              s(rule4) : "1:2:3=313"
              s(rule2) : "2:3=223"
              s(rule1) : "1:2=212"
                s(cat) : "2"
                  s(x) : "x"
```

(*Continued on next page*)

```
. wgtcellcollapse candidate, var(x) cat(212)
. sreturn list
macros:
           s(goodrule) : "6"
              s(rule6) : "3:212=313"
                s(cat) : "212"
                  s(x) : "x"
. wgtcellcollapse candidate, var(x) cat(55)
. sreturn list
macros:
                s(cat) : "55"
                  s(x) : "x"
```

In the second call to the option `max(9)` was used to restrict the returned rules to the rules that deal with the original categories only (so rule 8 that involved a collapsed category 234 was omitted). In the third call, a list of rules that involve a collapsed category `cat(212)` was requested. Requests for nonexisting categories are not considered errors, but simply produce empty lists of "good rules".

4.5 Syntax of `wgtcellcollapse label`

`wgtcellcollapse label , variable(`*varname*`)` [`verbose force`]

`variable(`*varname*`)` is the collapsed variable to be labeled.

`verbose` outputs the labeling results. There may be a lot of output.

`force` instructs `wgtcellcollapse label` to only use categories present in the data.

An example is given in section 5.6 below.

4.6 Syntax of `wgtcellcollapse collapse`

`wgtcellcollapse collapse` [*if*] [*in*]`, variables(`*varlist*`) mincellsize(#)`
`saving(`*dofile_name*`)` [`generate(`*newvarname*`) replace append`
`feed(`*varname*`) strict sort(`*varlist*`) run maxpass(#) maxcategory(#)`
`zeroes(`*numlist*`) greedy`]

`variables(`*varlist*`)` provides the list of variables whose cells are to be collapsed. When more than one variable is specified, `wgtcellcollapse collapse` proceeds from right to left, i.e., first attempts to collapse the rightmost variable.

`mincellsize(#)` specifies the minimum cell size for the collapsed cells. For most weighting purposes, values of 30 to 50 can be recommended.

`generate(`*newvarname*`)` specifies the name of the collapsed variable to be created.

`feed(`*varname*`)` provides the name of an already existing collapsed variable.

`strict` modifies the behavior of `wgtcellcollapse collapse` so that only collapsing rules for which all participating categories have nonzero counts are utilized.

`sort(`*varlist*`)` sorts the data set before proceeding to collapse the cell. The default sort order is in terms of the values of the collapsed variable. A different sort order may produce a different set of collapsed cell when cells are tied on size.

`maxpass(`#`)` specifies the maximum number of passes through the data set. The default value is 10000.

`maxcategory(`#`)` is the maximum category value of the variable being collapsed. It is passed to the internal calls to `wgtcellcollapse candidate`, see above.

`zeroes(`*numlist*`)` provides a list of the categories of the collapsed variable that may have zero counts in the data.

`greedy` modifies the behavior `wgtcellcollapse collapse` to prefer the rules that collapse the maximum number of categories.

Options to deal with the do-file to write the collapsing code to:

`saving(`*dofile_name*`)` specifies the name of the do-file that will contain the cell collapsing code.

`replace` overwrites the do-file if one exists.

`append` appends the code to the existing do-file.

`run` specifies that the do-file created is run upon completion. This option is typically specified with most runs.

The primary intent of `wgtcellcollapse collapse` is to create the code that can be utilized in both a survey data file and a population targets data file that are assumed to have identically named variables. Thus it does not only manipulate the data in the memory and collapses the cells, but also produces the do-file code that can be recycled in automated weight production. To that effect, when a do-file is created with the `replace` and `saving()` options, the user needs to specify `generate()` option to provide the name of the collapsed variable; and when the said do-file is appended with the the `append` and `saving()` options, the name of that variable is provided with the `feed()` option.

The algorithm by which `wgtcellcollapse collapse` identifies the cells to be collapsed uses a variation of greedy search. It first identifies the cells with the lowest (positive) counts; finds the candidate rules for the variable(s) to be collapsed; evaluates the counts of the collapsed cells across all these candidate rules; and uses the rule that produces the smallest size of the collapsed cell across all applicable rules. So when it finds several rules that are applicable to the cell being currently processed that has a size of 5, and the candidate rules produce cells of sizes 7, 10 and 15, `wgtcellcollapse collapse` will use the rule that produces the cell of size 7. The algorithm runs until all cells have sizes of at least `mincellsize(`#`)` or until `maxpass(`#`)` passes through the data are executed. In real world situations with missing data, this basic algorithm often produces inconsistent results, generally because it fails to identify empty cells, or fully track the cells that have already been collapsed. For that reason, a number of options are provided to modify its behavior. Section 5 will demonstrate the typical failures, and the ways to overcome them.

Hint 1. Since `wgtcellcollapse collapse` works with the sample data, it will not be able to identify categories that are not observed in the sample (e.g., rare categories), but may be present in the population. This will lead to errors at the raking stage, when the control total matrices have more categories than the data, forcing `ipfraking` to stop. To help with that, the option `zeroes()` allows the user to pass the categories of the variables that are known to exist in the population but not in the sample.

Hint 2. The behavior of `wgtcellcollapse collapse, zeroes()` may still not be satisfactory. As it evaluates the sample sizes of the collapsed cells across a number of candidate rules that involve zero cells, it may pick up the rule with lowest number, and that rule may as well leave some other candidate rules with zero cells untouched. This creates problems when `wgtcellcollapse collapse` returns to those untouched cells, and looks for the existing cells to collapse them with, creating collapsing rules with breaks in the sequences. To improve upon that behavior, option `greedy` makes `wgtcellcollapse collapse` look for a rule that has as many categories as possible, thus collapsing as many categories with zero counts in one pass as it can.

Hint 3. Other than for dealing with zero cells, the option `strict` should be specified most of the time. It effectively makes sure that the candidate rules correspond to the actual data.

Hint 4. If you want to guarantee some specific combination of cells to be collapsed by `wgtcellcollapse collapse`, the most reliable way is to explicitly identify them with the *if* condition, and specify a very large cell size like `mincellsize(10000)` so that `wgtcellcollapse collapse` makes every possible effort to collapse those cells. Since the resulting cell(s) will fall short of that size, the program will exit with a complaint that this size could not be achieved, but hopefully the cells will be collapsed as needed.

A referee noted that `wgtcellcollapse` could also have utility in preparing for hotdeck imputation procedures. The textbook versions of hotdeck procedures impute missing data by assuming a MAR model (Rubin 1976) with conditioning on a set of categorical variables, i.e., cells of a multivariate table. As is in weighting procedures, hotdeck procedures are more stable with larger cells, so cell collapsing is often recommended to achieve minimal cell sizes (with an understanding of the bias-vs-variance tradeoff built into these collapsing decisions). For a review of the hotdeck and related imputation methods, see Andridge and Little (2010).

5 Extended motivating example

The primary purpose of developing `wgtcellcollapse` and adding it to the `ipfraking` suite was to address the need to collapse cells of the margin variables so that each cell has a minimum sample size; and to do so in a way that can be easily made consistent between a sample data and the population targets data. The problem arises when some of the target variables have dozens of categories, most of which have small counts. Example where such needs arise include:

- transportation surveys, where many stations will have low counts of boardings and especially alightings;
- country of origin variables in household surveys, where most countries will have very low counts;
- continuous age variables which can be collapsed into age groups differently for the different races.

A referee pointed out that an identical problem arises in creating cells for hot deck imputation procedures Andridge and Little (2010), so `wgtcellcollapse` could also have utility in that application.

The workflow of `wgtcellcollapse` is demonstrated with the following simulated transportation data set of trips along a commuter metro line composed of 21 stations:

```
. use stations, clear
. list station_id, sep(0)

     +---------------------+
     |          station_id |
     |---------------------|
  1. |          1. Alewife |
  2. |        2. Brookline |
  3. |        8. Carmenton |
  4. |        11. Dogville |
  5. |        18. East End |
  6. |      24. Framington |
  7. |   26. Grand Junction |
  8. |      30. High Point |
  9. |      36. Irvingtown |
 10. |      39. Johnsville |
 11. |     40. King Street |
 12. |        44. Limerick |
 13. |     47. Moscow City |
 14. |    49. Ninth Street |
 15. |    50. Ontario Lake |
 16. | 53. Picadilly Square |
 17. |      55. Queens Zoo |
 18. |   60. Redline Circle |
 19. |    62. Silver Spring |
 20. |     68. Toledo Town |
 21. |    69. Union Station |
     +---------------------+
```

Suppose turnstile counts were collected at entrances and exits of these stations, producing the following population figures.

(*Continued on next page*)

```
. use trip_population, clear
. table board_id daypart , c(sum num_pass) cellwidth(10) mi
```

board_id	daypart AM Peak	Midday	PM Reverse	Night	Weekend
1. Alewife	1423	34	219	113	44
2. Brookline	7198	298	773	169	144
8. Carmenton	19254	181	3739	872	422
11. Dogville	12626	872	3476	769	1270
18. East End	2470	143	1263	145	114
24. Framington	634	50	1296	133	60
26. Grand Junction	2208	233	439	88	166
30. High Point	4319	424	3740	482	115
36. Irvingtown	1221	34	444	30	167
39. Johnsville	93	4	64	2	6
40. King Street	398	46	76	11	13
44. Limerick	1021	19	129	53	34
47. Moscow City	3300	776	984	140	301
49. Ninth Street	38	22	191	5	5
50. Ontario Lake	606	22	80	18	23
53. Picadilly Square	642	71	622	153	69
55. Queens Zoo	331	23	174	15	19
60. Redline Circle	270	4	63	13	3
62. Silver Spring	3402	240	950	206	445
68. Toledo Town	5085	61	744	272	112

```
. table alight_id daypart , c(sum num_pass) cellwidth(10) mi
```

alight_id	daypart AM Peak	Midday	PM Reverse	Night	Weekend
2. Brookline	19	.	3	2	.
8. Carmenton	492	18	56	23	15
11. Dogville	2475	42	423	153	80
18. East End	929	31	193	67	68
24. Framington	404	13	91	28	27
26. Grand Junction	576	20	147	42	41
30. High Point	2189	89	560	165	167
36. Irvingtown	288	10	91	21	18
39. Johnsville	41	.	11	2	1
40. King Street	131	3	38	8	6
44. Limerick	277	9	87	20	18
47. Moscow City	1746	78	556	142	128
49. Ninth Street	88	2	25	3	4
50. Ontario Lake	232	11	70	14	14
53. Picadilly Square	633	33	198	47	47
55. Queens Zoo	230	10	71	13	14
60. Redline Circle	90	2	26	3	4
62. Silver Spring	1134	67	369	91	85
68. Toledo Town	1372	81	444	112	118
69. Union Station	53193	3038	16007	2733	2677

Most people ride the train to the last station, with much smaller traffic at other population centers.

Suppose a survey was administered to a sample of the metro line users, with the following counts of cases collected.

```
. use trip_sample, clear
. table board_id daypart , c(freq) cellwidth(10) mi
```

board_id	daypart AM Peak	Midday	PM Reverse	Night	Weekend
1. Alewife	46	4	11	7	3
2. Brookline	236	4	35	6	7
8. Carmenton	653	4	184	47	24
11. Dogville	410	41	166	35	56
18. East End	85	5	64	4	4
24. Framington	30	3	74	3	1
26. Grand Junction	72	13	23	5	6
30. High Point	158	20	187	25	12
36. Irvingtown	34	2	25	1	15
39. Johnsville	5	1	1	.	.
40. King Street	17	1	2	.	1
44. Limerick	28	.	9	1	3
47. Moscow City	94	31	49	7	13
49. Ninth Street	.	.	9	.	.
50. Ontario Lake	13	1	4	1	1
53. Picadilly Square	23	4	35	7	5
55. Queens Zoo	10	1	14	.	2
60. Redline Circle	13	.	5	.	.
62. Silver Spring	106	18	38	12	17
68. Toledo Town	149	6	33	11	3

```
. table alight_id daypart , c(freq) cellwidth(10) mi
```

alight_id	daypart AM Peak	Midday	PM Reverse	Night	Weekend
2. Brookline	1	.	.	.	.
8. Carmenton	11	1	1	.	1
11. Dogville	85	1	14	6	5
18. East End	36	1	18	1	4
24. Framington	15	1	2	2	2
26. Grand Junction	15	2	8	1	1
30. High Point	73	4	22	11	8
36. Irvingtown	9	.	4	2	2
39. Johnsville	3	.	1	.	.
40. King Street	.	.	3	.	.
44. Limerick	13	.	2	.	2
47. Moscow City	81	6	22	6	6
49. Ninth Street	3	1	1	.	.
50. Ontario Lake	2	.	1	2	1
53. Picadilly Square	23	1	8	3	2
55. Queens Zoo	6	.	5	1	.
60. Redline Circle	5	.	.	.	.
62. Silver Spring	49	.	19	3	9
68. Toledo Town	43	3	24	6	7
69. Union Station	1,709	138	813	128	123

As only 3654 surveys were collected from a total of 96783 riders, we would reasonably expect that there is a need for weighting and nonresponse adjustment. The data available for calibration includes the population turnstile counts listed above. We will produce interactions of the day part and the station that will serve as two weighting margins (one for the stations where the metro users boarded, and one for the stations where they got off).

First, we need to define the weighting rules. In this case, the stations are numbered sequentially, with the northernmost, say, station Alewife being number 3, and the southernmost station, Union Station, where everybody gets off to rush to their city jobs or attractions, being number 69. Below, we create a list of stations and provide it to `wgtcellcollapse sequence`. We would be collapsing stations along the line, with the expectation that travelers boarding or leaving at adjacent stations within the same day part are more similar to one another than the travelers boarding or leaving a particular station at different times of the day. Collapsing rules need to be defined for the `daypart` variable as well — mostly because `wgtcellcollapse collapse` expects all variables to have collapsing rules defined.

```
. use trip_sample, clear
. wgtcellcollapse sequence , var(daypart) from(2 3 4) depth(3)
. levelsof board_id, local(stations_on)
1 2 8 11 18 24 26 30 36 39 40 44 47 49 50 53 55 60 62 68
. levelsof alight_id, local(stations_off)
2 8 11 18 24 26 30 36 39 40 44 47 49 50 53 55 60 62 68 69
. local all_stations : list stations_on | stations_off
. * relies on stations being in sequential order!!!
. wgtcellcollapse sequence , var(board_id alight_id) from(`all_stations´) depth(20)
. save trip_sample_rules, replace
file trip_sample_rules.dta saved
```

The syntax above relies on the stations being in the sequential order, which is how the output of `levelsof` is organized. Otherwise, the internal numeric identifiers of the stations would need to be supplied in the order in which the trains run through them.

The number of collapsing rules for variables `board_id` and `alight_id` created by `wgtcellcollapse sequence` is 2961 each.

5.1 The first pass of cell collapse and raking

Let us say that we want to define weighting cells with at least 20 cases in each. We will thus start with weighting cells defined as station-by-daypart interaction, and collapsing stations within daypart to achieve the cell sizes of at least 20 cases. Here is what a simple run of `wgtcellcollapse collapse` might look like.

(*Continued on next page*)

```
. use trip_sample_rules, clear
. wgtcellcollapse collapse, variables(daypart board_id) mincellsize(20) ///
>         generate(dpston1) saving(dpston1.do) replace run
Pass 0 through the data...
  smallest count = 1 in the cell      2000039
  Invoking rule 39:40=23940
  replace dpston1 = 2023940 if inlist(dpston1, 2000039, 2000040)
Pass 1 through the data...
  smallest count = 1 in the cell      2000050
  Invoking rule 50:53=25053
  replace dpston1 = 2025053 if inlist(dpston1, 2000050, 2000053)
Pass 2 through the data...
  smallest count = 1 in the cell      2000055
  Invoking rule 55:25053=35055
  replace dpston1 = 2035055 if inlist(dpston1, 2000055, 2025053)
Pass 3 through the data...
  smallest count = 1 in the cell      3000039
  Invoking rule 39:40=23940
  replace dpston1 = 3023940 if inlist(dpston1, 3000039, 3000040)
  (output omitted)
Pass 35 through the data...
  smallest count = 11 in the cell      5031826
  Invoking rule 30:31826=41830
  replace dpston1 = 5041830 if inlist(dpston1, 5000030, 5031826)
Pass 36 through the data...
  smallest count = 12 in the cell      2065068
  WARNING: could not find any rules to collapse dpston1 == 2065068
Pass 37 through the data...
  smallest count = 12 in the cell      3033944
  Invoking rule 26:23036:33944=62644
  replace dpston1 = 3062644 if inlist(dpston1, 3000026, 3023036, 3033944)
  (output omitted)
Pass 38 through the data...
  smallest count = 13 in the cell      1000050
  Invoking rule 50:53=25053
  replace dpston1 = 1025053 if inlist(dpston1, 1000050, 1000053)
  (output omitted)
Pass 43 through the data...
  smallest count = 14 in the cell      3000055
  Invoking rule 53:55=25355
  replace dpston1 = 3025355 if inlist(dpston1, 3000053, 3000055)
Pass 44 through the data...
  smallest count = 15 in the cell      5104068
  WARNING: could not find any rules to collapse dpston1 == 5104068
Pass 45 through the data...
  smallest count = 17 in the cell      5000062
  Invoking rule 11:18:24:26:30:36:39:40:44:47:49:50:53:55:60:62=161162
  replace dpston1 = 5161162 if inlist(dpston1, 5000011, 5000018, 5000024, 5000026, 5
> 000030, 5000036, 5000039, 5000040, 5000044, 5000047, 5000049, 5000050, 5000053, 50
> 00055, 5000060, 5000062)
Pass 46 through the data...
  smallest count = 18 in the cell      2000062
  Invoking rule 30:36:39:40:44:47:49:50:53:55:60:62=123062
  replace dpston1 = 2123062 if inlist(dpston1, 2000030, 2000036, 2000039, 2000040, 2
> 000044, 2000047, 2000049, 2000050, 2000053, 2000055, 2000060, 2000062)
Pass 47 through the data...
  smallest count = 18 in the cell      3054960
  Invoking rule 62:54960=64962
```

```
  replace dpston1 = 3064962 if inlist(dpston1, 3000062, 3054960)
Pass 48 through the data...
  smallest count = 22 in the cell      1023940
  Done collapsing! Exiting...
. return list
scalars:
         r(arg_min_id) =  1023940
                r(min) =  22
macros:
            r(cfailed) : "2065068,5104068"
             r(failed) : "2065068 5104068"
. wgtcellcollapse collapse, variables(daypart alight_id) mincellsize(20) ///
>         generate(dpstoff1) saving(dpstoff1.do) replace run
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 2:8=20208
  replace dpstoff1 = 1020208 if inlist(dpstoff1, 1000002, 1000008)
Pass 1 through the data...
  smallest count = 1 in the cell      2000008
  Invoking rule 8:11=20811
  replace dpstoff1 = 2020811 if inlist(dpstoff1, 2000008, 2000011)
  (output omitted)
Pass 53 through the data...
  smallest count = 16 in the cell      5026268
  Invoking rule 69:26268=36269
  replace dpstoff1 = 5036269 if inlist(dpstoff1, 5000069, 5026268)
Pass 54 through the data...
  smallest count = 22 in the cell      3000047
  Done collapsing! Exiting...
. return list
scalars:
         r(arg_min_id) =  3000047
                r(min) =  22
macros:
            r(cfailed) : "2044753,4055368"
             r(failed) : "2044753 4055368"
```

Each pass identified the smallest cell count, the cell where this low count is found, the rule that can be used to collapse this cell with some other cell (see more on determination of what `wgtcellcollapse` believes to be the best rule below), and Stata code that can be used to apply this collapsing rule.

The collapsed values of the variables `dpston` (DayPart-STation-ON) and `dpstoff` (DayPart-STation-OFF) combine the values of the parent variables. The value of `dpston==1000003` indicates the combination of categories `daypart==1` and station number 3. The value of `dpston==2023940`, the very first collapsed cell, indicates `daypart==2` and sequence of two stations from 39 to 40. The value of `dpston==3064960` indicates `daypart==3` and sequence of six stations from 49 to 60.

Note that in passes 36 and 44 for the boarding counts, `wgtcellcollapse` could not find any applicable rules to collapse the cells and improve the count, of which a warning was issued in the output. Additionally, the problematic value are returned to the user in the `r(failed)` macro as a space delimited list, and in the `r(cfailed)`, as a commadelimited list. The content of the `r(failed)` macro can be used in code that could read

```
foreach c in `r(failed)´ {
    ...
    * run some diagnostics for each category that failed
    ...
}
```

while the content of the the `r(cfailed)` macro can be used in code that could read

```
list ... if inlist(dpston,`r(cfailed)´)
```

Also, these returned values should be used in production code by using the `assert` command (Gould 2003) to ascertain that these macros are empty (i.e., no errors were encountered):

```
assert "`r(failed)´" == ""
```

In Pass 48, `wgtcellcollapse` finds that, with the exception of the failures noted above, all successfully collapsed cells now have sizes above the required minimum (option `mincellsize(20)`), and exits.

Similar output is produced for collapsing of the cells for alighting counts. Only a few lines of output, out of several hundred, are shown.

While we know that some cell counts are less than 20, we will ignore the issue for the moment, as there are bigger concerns with the collapsed cells, as will become clear once we follow through with the workflow of weighting, and attempt raking.

From the above run, `wgtcellcollapse` produced two files, one for each weighting margin, called `dpston.do` and `dpstoff.do`. An interested reader is welcome to `list` them; they contain long sequences of `replace` commands to perform the cell collapsing. These do-files can be run on the population data to create identical categories and produce the matrices of the population control totals for `ipfraking` to use:

```
. use trip_population, clear
. run dpston1.do
. total num_pass , over(dpston1)
Total estimation                  Number of obs   =        719
      1000001: dpston1 = 1000001
      1000002: dpston1 = 1000002
  (output omitted)
      5104068: dpston1 = 5104068
      5161162: dpston1 = 5161162
--------------------------------------------------------------
         Over |      Total   Std. Err.     [95% Conf. Interval]
--------------+-----------------------------------------------
num_pass      |
      1000001 |       1423   967.7508     -476.9595    3322.959
      1000002 |       7198    4895.91     -2414.011    16810.01
  (output omitted)
      5104068 |        270   116.7702      40.74822    499.2518
```

```
    5161162 |       1723   909.6551     -62.90172    3508.902
-----------------------------------------------------------

. matrix dpston1 = e(b)

. matrix coleq dpston1 = _one

. matrix rownames dpston1 = dpston1

. run dpstoff1.do

. total num_pass , over(dpstoff1)

Total estimation                  Number of obs   =        719

      1000011: dpstoff1 = 1000011
      1000018: dpstoff1 = 1000018
  (output omitted)
      5000060: dpstoff1 = 5000060
      5036269: dpstoff1 = 5036269
      5140853: dpstoff1 = 5140853

-----------------------------------------------------------
       Over |      Total   Std. Err.     [95% Conf. Interval]
-------------+---------------------------------------------
num_pass    |
    1000011 |       2475   1468.807     -408.6691    5358.669
    1000018 |        929   360.7303      220.7878    1637.212
  (output omitted)
    5000060 |          4          2      .0734531    7.926547
    5036269 |       2880   980.8909      954.2428    4805.757
    5140853 |        634   139.2172      360.6787    907.3213
-----------------------------------------------------------

. matrix dpstoff1 = e(b)

. matrix coleq dpstoff1 = _one

. matrix rownames dpstoff1 = dpstoff1
```

We can then go back to the sample data and try creating raked weights:

```
. use trip_sample, clear

. run dpston1

. run dpstoff1

. gen byte _one = 1

. ipfraking [pw=_one], ctotal(dpston1 dpstoff1) gen(raked_weight1)
categories of dpston1 do not match in the control dpston1 and in the data (nolab opt
> ion)
This is what dpston1 gives:
  _one:1000001 _one:1000002 _one:1000008 _one:1000011 _one:1000018 _one:1000024 _one
> :1000026 _one:1000030 _one:1000036 _one:1000044 _one:1000047 _one:1000049 _one:100
> 0062 _one:1000068 _one:1023940 _one:1025053 _one:1025560 _one:2000011 _one:2065068
>  _one:2070126 _one:2110847 _one:2123062 _one:3000008 _one:3000011 _one:3000018 _on
> e:3000024 _one:3000030 _one:3000036 _one:3000047 _one:3000068 _one:3020102 _one:30
> 25355 _one:3062644 _one:3064962 _one:4030108 _one:4041830 _one:4084768 _one:413115
> 3 _one:5030108 _one:5041830 _one:5053647 _one:5104068 _one:5161162
This is what I found in data:
  _one:1000001 _one:1000002 _one:1000008 _one:1000011 _one:1000018 _one:1000024 _one
> :1000026 _one:1000030 _one:1000036 _one:1000044 _one:1000047 _one:1000062 _one:100
> 0068 _one:1023940 _one:1025053 _one:1025560 _one:2000011 _one:2065068 _one:2070126
>  _one:2110847 _one:2123062 _one:3000008 _one:3000011 _one:3000018 _one:3000024 _on
> e:3000030 _one:3000036 _one:3000047 _one:3000068 _one:3020102 _one:3025355 _one:30
```

```
> 62644 _one:3064962 _one:4030108 _one:4041830 _one:4084768 _one:4131153 _one:503010
> 8 _one:5041830 _one:5053647 _one:5104068 _one:5161162
This is what dpston1 has that data don´t:
  _one:1000049
This is what data have that dpston1 doesn´t:
  <none>
r(111);
. ipfraking [pw=_one], ctotal(dpstoff1 dpston1) gen(raked_weight1)
categories of dpstoff1 do not match in the control dpstoff1 and in the data (nolab o
> ption)
This is what dpstoff1 gives:
  _one:1000011 _one:1000018 _one:1000030 _one:1000047 _one:1000068 _one:1000069 _one
> :1025355 _one:1043644 _one:1060226 _one:1064962 _one:2044753 _one:2190869 _one:300
> 0002 _one:3000047 _one:3000068 _one:3000069 _one:3050826 _one:3053044 _one:3064962
>  _one:4000002 _one:4000008 _one:4000049 _one:4055368 _one:4075069 _one:4101147 _on
> e:5000055 _one:5000060 _one:5036269 _one:5140853
This is what I found in data:
  _one:1000011 _one:1000018 _one:1000030 _one:1000047 _one:1000068 _one:1000069 _one
> :1025355 _one:1043644 _one:1060226 _one:1064962 _one:2044753 _one:2190869 _one:300
> 0047 _one:3000068 _one:3000069 _one:3050826 _one:3053044 _one:3064962 _one:4055368
>  _one:4075069 _one:4101147 _one:5036269 _one:5140853
This is what dpstoff1 has that data don´t:
  _one:3000002 _one:4000002 _one:4000008 _one:4000049 _one:5000055 _one:5000060
This is what data have that dpstoff1 doesn´t:
  <none>
r(111);
.
```

We see that raking failed, because survey nonresponse wiped out some of the smaller stations from the sample. (Note also the informative error message with diagnostics of missing categories produced by `ipfraking`. This is a functionality added since Kolenikov (2014). The message lists the categories found in the data, in the control totals, and in the mismatch.) We may have suspected as much from the full output of the population control totals. For instance, the line in the output of the `total` command for `dpstoff==5000060` showed only 4 alightings at station Redline Circle (60) on the weekends, a much lower count than others.

5.2 The second pass of cell collapse and raking: `zeroes()` option

Having identified the issue, we can overcome it with the `zeroes()` option of `wgtcellcollapse collapse` whose purpose is specifically to add missing categories. This option provides the list of stations that may have zero sample counts in a given daypart. For instance, notice that the sample registers only one alighting at Brookline (2) in AM Peak daypart, even though there are passengers exiting in other dayparts. All in all, `wgtcellcollapse` needs to be made aware of the zero sample boardings at Johnsville (39), King Street (40), Limerick (44), Ninth Street (49), Queens Zoo (55) and Redline Circle (60); as well as zero alightings at Brookline (2), Carmenton (8), Irvingtown (36), Johnsville (39), King Street (40), Limerick (44), Moscow City (47), Ninth Street (49), Ontario Lake (50), Queens Zoo (55), Redline Circle (60), and Silver Spring (62).

(*Continued on next page*)

```
. use trip_sample_rules, clear
. wgtcellcollapse collapse, variables(daypart board_id) mincellsize(20) ///
>         zeroes(39 40 44 49 55 60) ///
>         generate(dpston2) saving(dpston2.do) replace run
Pass 0 through the data...
  smallest count = 1 in the cell      2000039
Processing zero cells...
  Invoking rule 49:50=24950 to collapse zero cells
  replace dpston2 = 1024950 if inlist(dpston2, 1000049, 1000050)
Pass 0 through the data...
  smallest count = 1 in the cell      2000039
  Invoking rule 40:44=24044 to collapse zero cells
  replace dpston2 = 2024044 if inlist(dpston2, 2000040, 2000044)
  (output omitted)
Pass 0 through the data...
  smallest count = 1 in the cell      2000039
  Invoking rule 24950:25355:60=54960 to collapse zero cells
  replace dpston2 = 5054960 if inlist(dpston2, 5024950, 5025355, 5000060)
Pass 0 through the data...
  smallest count = 1 in the cell      2000039
Pass 12 through the data...
  smallest count = 1 in the cell      2000039
  Invoking rule 39:24044=33944
  replace dpston2 = 2033944 if inlist(dpston2, 2000039, 2024044)
  (output omitted)
Pass 58 through the data...
  smallest count = 18 in the cell      4055368
  WARNING: could not find any rules to collapse dpston2 == 4055368
Pass 59 through the data...
  smallest count = 20 in the cell      2000030
  Done collapsing! Exiting...
. return list
scalars:
        r(arg_min_id) =  2000030
               r(min) =  20
macros:
           r(cfailed) : "5055368,4055368"
            r(failed) : "5055368 4055368"
. wgtcellcollapse collapse, variables(daypart alight_id) mincellsize(20) ///
>         zeroes(2 8 36 39 40 44 47 49 50 55 60 62) ///
>         generate(dpstoff2) saving(dpstoff2.do) replace run
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
Processing zero cells...
  Invoking rule 39:40=23940 to collapse zero cells
  replace dpstoff2 = 1023940 if inlist(dpstoff2, 1000039, 1000040)
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 2:8=20208 to collapse zero cells
  replace dpstoff2 = 2020208 if inlist(dpstoff2, 2000002, 2000008)
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 30:36=23036 to collapse zero cells
  replace dpstoff2 = 2023036 if inlist(dpstoff2, 2000030, 2000036)
  (output omitted)
```

```
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 24950:53:55:60=54960 to collapse zero cells
  replace dpstoff2 = 5054960 if inlist(dpstoff2, 5024950, 5000053, 5000055, 5000060)
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
Pass 24 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 2:8=20208
  replace dpstoff2 = 1020208 if inlist(dpstoff2, 1000002, 1000008)
  (output omitted)
Pass 38 through the data...
  smallest count = 2 in the cell      1000050
  Invoking rule 49:50=24950
  replace dpstoff2 = 1024950 if inlist(dpstoff2, 1000049, 1000050)
  (output omitted)
Pass 47 through the data...
  smallest count = 3 in the cell      1023940
  Invoking rule 36:23940=33640
  replace dpstoff2 = 1033640 if inlist(dpstoff2, 1000036, 1023940)
  (output omitted)
Pass 55 through the data...
  smallest count = 5 in the cell      1000060
  Invoking rule 24950:25355:60=54960
  replace dpstoff2 = 1054960 if inlist(dpstoff2, 1024950, 1025355, 1000060)
  (output omitted)
Pass 76 through the data...
  smallest count = 15 in the cell      3054960
  Invoking rule 62:54960=64962
  replace dpstoff2 = 3064962 if inlist(dpstoff2, 3000062, 3054960)
Pass 77 through the data...
  smallest count = 21 in the cell      4070230
  Done collapsing! Exiting...
. return list
scalars:
         r(arg_min_id) =  4070230
                r(min) =  21
macros:
            r(cfailed) : "2023036,2110244,5103660"
             r(failed) : "2023036 2110244 5103660"
```

Note that `wgtcellcollapse` reports specifically that it processes zero cells, and then proceeds to the nonempty cells at Pass 12 for boarding, and at Pass 24 for alightings.

We will continue to disregard the cell counts of insufficient size for the time being. Running the resulting do-files `dpston.do` and `dpstoff.do` on the population data to create control totals, and providing these control totals to `ipfraking` program produces an apparently successful raking result:

```
. use trip_sample, clear
. run dpston2
. run dpstoff2
. gen byte _one = 1
```

```
. ipfraking [pw=_one], ctotal(dpston2 dpstoff2) gen(raked_weight2)
 Iteration 1, max rel difference of raked weights = 36.208881
 Iteration 2, max rel difference of raked weights = .05484732
 Iteration 3, max rel difference of raked weights = .0055794
 Iteration 4, max rel difference of raked weights = .00053851
 Iteration 5, max rel difference of raked weights = .00005171
 Iteration 6, max rel difference of raked weights = 4.962e-06
 Iteration 7, max rel difference of raked weights = 4.762e-07
The worst relative discrepancy of  3.9e-08 is observed for dpstoff2 == 5180262
Target value =        483; achieved value =        483
   Summary of the weight changes
```

	Mean	Std. dev.	Min	Max	CV
Orig weights	1	0	1	1	0
Raked weights	26.487	5.9013	8.1096	37.001	.2228
Adjust factor	26.4869		8.1096	37.0014	

```
. whatsdeff raked_weight2
```

Group	Min	Mean	Max	CV	DEFF	N	N eff
Overall	8.11	26.49	37.00	0.2228	1.0496	3654	3481.24

Note the use of utility program `whatsdeff` to compute the design effect due to unequal weighting; see section 2.4. The problem of zero cells appeared to have been solved: each and every population combination of `daypart` and `station` is properly reflected in control total categories, and there are no error messages concerning mismatching categories.

The weighting cells, however, are still not without problems. Consider this cross-tab of original and collapsed stations:

```
. tab alight_id dpstoff2 if daypart == 1 & mod(dpstoff2,100*100)>99
```

	Long ID of the interaction				
alight_id	1025355	1043644	1060226	1064962	Total
2. Brookline	0	0	1	0	1
8. Carmenton	0	0	11	0	11
24. Framington	0	0	15	0	15
26. Grand Junction	0	0	15	0	15
36. Irvingtown	0	9	0	0	9
39. Johnsville	0	3	0	0	3
44. Limerick	0	13	0	0	13
49. Ninth Street	0	0	0	3	3
50. Ontario Lake	0	0	0	2	2
53. Picadilly Square	23	0	0	0	23
55. Queens Zoo	6	0	0	0	6
60. Redline Circle	0	0	0	5	5
62. Silver Spring	0	0	0	49	49
Total	29	25	42	59	155

Here, the first part of the *if* expression identifies the AM peak. The second part identifies collapsed stations, given the nomenclature of `dpstoff` variable described on page 5.1. The collapsed category code is a the concatenation of the value of the first variable of the interaction, `daypart`; the length of the collapsed sequence; and its starting and end points. Station numbers take up to two characters, and hence the collapsed values would use categories of `alight_id` like 20102. Hence collapsed cells could be identified as `mod` by `100*100` being greater than the maximum two-digit number, 99.

In the output, it appears that Picadilly Square (53) and Queens Zoo (55) should have been a part of the six-station sequence 1064962 spanning from Ninth Street (49) to Silver Spring (62). Instead, `wgtcellcollapse` decided to separate these two stations out into their own cell. How did that happen? The logic of `wgtcellcollapse` is to collapse categories in such a way as to produce the result with the smallest possible count. Thus, within AM peak day part, the sequence of collapsing steps was as follows (the numbers refer to the full output):

Pass 0 The zero cells were collapsed first: Johnsville (39) and King Street (40) resulting in an intermediate cell of size 3.

Pass 24 The smallest cell of size 1 (Brookline (2)) was collapsed with its neighbor (Carmenton (8)) resulting in an intermediate cell of size 12.

Pass 38 The smallest cell of size 2 (Ontario Lake (50)) was collapsed with its neighbor (Ninth Street (49)) resulting in an intermediate cell of size 5.

Pass 47 The smallest cell of size 3, collapsed Johnsville (39) + King Street (40), was further collapsed with its neighbor Irvingtown (36) resulting in an intermediate cell of size 12.

Pass 55 The smallest cell of size 5, Redline Circle (60), was collapsed by a three-way rule with a duo Picadilly Square (53) + Queens Zoo (55), which actually was empty, and a small cell Ontario Lake (50) + Ninth Street (49), resulting in an intermediate cell of size 10.

Let us look at that last step in more detail. At this stage, Redline Circle (60) with 5 exiting passengers in the sample could be collapsed with:

1. Silver Spring (62), to form a cell of size 54;
2. Queens Zoo (55), to form a cell of size 11;
3. a sequence of Picadilly Square (53) and Queens Zoo (55), to form a cell of size 34;
4. ... and a number of other options

However, at pass 55, `wgtcellcollapse` picked the rule 24950:25355:60=54960 which, at the time it was processed, had a count of 5 in the cell 24950, a count of zero in the cell 25355, and a count of 5 in the original station Redline Circle (60). (Note that the cell 25355 would actually form later at pass 58.) This produced the smallest count of the resulting cell, which according to the minimalist logic of `wgtcellcollapse` is the best route to go.

The problem lies with the zero count of the ghost of the cell 25355, and the application of a rule that contains this ghost cell!

To overcome this problem, `wgtcellcollapse` using `strict` option would only allow for the rules that have a non-zero count in every component of the rule (so the problematic rule 24950:25355:60=54960 would not be a legal one under that restriction). As is easily seen, this option directly contradicts the `zeroes()` option, and that necessitates separate runs.

5.3 The third pass of cell collapse and raking: `strict` and `feed` options

We will separate the two runs of `wgtcellcollapse` into a run that only deals with zeroes, and another run that deals with everything else. To prevent `wgtcellcollapse` from any further merges, `mincellsize(1)` can be specified in the first run. As the relevant variables will have already been created by the first run, the option to pass the variable name to be further modified is `feed()`. To make sure that the relevant variable exists in the data set, the option `run` instructs `wgtcellcollapse` to run the do-file it just created, thus creating or modifying the collapsed cell variable. Finally, instead of specifying `replace` to overwrite the do-files that `wgtcellcollapse` creates, we need to specify `append` to keep adding to these files.

```
. use trip_sample_rules, clear
. wgtcellcollapse collapse, variables(daypart board_id) mincellsize(1) ///
>         zeroes(39 40 44 49 55 60) ///
>         generate(dpston3) saving(dpston3.do) replace run
Pass 0 through the data...
  smallest count = 1 in the cell      2000039
Processing zero cells...
  Invoking rule 49:50=24950 to collapse zero cells
  replace dpston3 = 1024950 if inlist(dpston3, 1000049, 1000050)
Pass 0 through the data...
  smallest count = 1 in the cell      2000039
  Invoking rule 40:44=24044 to collapse zero cells
  replace dpston3 = 2024044 if inlist(dpston3, 2000040, 2000044)
  (output omitted)
Pass 0 through the data...
  smallest count = 1 in the cell      2000039
  Invoking rule 24950:25355:60=54960 to collapse zero cells
  replace dpston3 = 5054960 if inlist(dpston3, 5024950, 5025355, 5000060)
Pass 0 through the data...
  smallest count = 1 in the cell      2000039
Pass 12 through the data...
  smallest count = 1 in the cell      2000039
  Done collapsing! Exiting...
. wgtcellcollapse collapse, variables(daypart board_id) mincellsize(20) ///
>         strict feed(dpston3) saving(dpston3.do) append run
Pass 12 through the data...
  smallest count = 1 in the cell      2000039
  Invoking rule 39:24044=33944
  replace dpston3 = 2033944 if inlist(dpston3, 2000039, 2024044)
```

(*output omitted*)

```
Pass 57 through the data...
  smallest count = 19 in the cell      3025560
  Invoking rule 62:25560=35562
  replace dpston3 = 3035562 if inlist(dpston3, 3000062, 3025560)
Pass 58 through the data...
  smallest count = 20 in the cell      5026268
  Done collapsing! Exiting...
. wgtcellcollapse collapse, variables(daypart alight_id) mincellsize(1) ///
>        zeroes(2 8 36 39 40 44 47 49 50 55 60 62) ///
>        generate(dpstoff3) saving(dpstoff3.do) replace run
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
Processing zero cells...
  Invoking rule 39:40=23940 to collapse zero cells
  replace dpstoff3 = 1023940 if inlist(dpstoff3, 1000039, 1000040)
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 2:8=20208 to collapse zero cells
  replace dpstoff3 = 2020208 if inlist(dpstoff3, 2000002, 2000008)
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 30:36=23036 to collapse zero cells
  replace dpstoff3 = 2023036 if inlist(dpstoff3, 2000030, 2000036)
```

(*output omitted*)

```
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 24950:53:55:60=54960 to collapse zero cells
  replace dpstoff3 = 5054960 if inlist(dpstoff3, 5024950, 5000053, 5000055, 5000060)
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
Pass 24 through the data...
  smallest count = 1 in the cell      1000002
  Done collapsing! Exiting...
. wgtcellcollapse collapse, variables(daypart alight_id) mincellsize(20) ///
>        strict feed(dpstoff3) saving(dpstoff3.do) append run
Pass 24 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 2:8=20208
  replace dpstoff3 = 1020208 if inlist(dpstoff3, 1000002, 1000008)
Pass 25 through the data...
  smallest count = 1 in the cell      2000011
  Invoking rule 11:18=21118
  replace dpstoff3 = 2021118 if inlist(dpstoff3, 2000011, 2000018)
```

(*output omitted*)

```
Pass 76 through the data...
  smallest count = 16 in the cell      5026268
  Invoking rule 170260:26268=190268
  replace dpstoff3 = 5190268 if inlist(dpstoff3, 5170260, 5026268)
Pass 77 through the data...
  smallest count = 21 in the cell      4070230
  Done collapsing! Exiting...
```

The result still isn't satisfactory, as some collapsed cells still overlap:

```
. tab alight_id dpstoff3 if daypart == 2 & mod(dpstoff3,100*100)>99
```

	Long ID of the interaction			
alight_id	2023036	2042639	2200269	Total
8. Carmenton	0	0	1	1
11. Dogville	0	0	1	1
18. East End	0	0	1	1
24. Framington	0	0	1	1
26. Grand Junction	0	2	0	2
30. High Point	4	0	0	4
47. Moscow City	0	0	6	6
49. Ninth Street	0	0	1	1
53. Picadilly Square	0	0	1	1
68. Toledo Town	0	0	3	3
69. Union Station	0	0	138	138
Total	4	2	153	159

This overlap can be traced back to the collapsing of zero cells: first, the cell 2023036 came to being by a reasonable, at its face, collapsing of the zero cell Irvingtown (36) with non-zero cell High Point (30); and then the cell 2042639 came to being by a long overreach for the zero cell Johnsville (39) to be collapsed with a non-zero cell Grand Junction (26). The resulting cells can neither be collapsed together, nor added to the "everything" cell 2200269.

5.4 The fourth pass of cell collapse and raking: `greedy` and `maxcat()` options

The next improvement is to fix the problem of weak performance in collapsing the zero cells with an additional option `greedy`. It modifies the behavior of `wgtcellcollapse` to require that, among the possible candidate rules with the lowest count, the rule with the *greatest* number of components is preferred. That way, the long streaks of zeroes from Irvingtown (36) to Limerick (44) in the midday part could be collapsed simultaneously into one cell. To support this option, and avoid complex collapses of zero cells with the already defined cells, option `maxcategory()` specifies the greatest value of a component of a rule. By specifying `maxcategory(99)`, we can instruct `wgtcellcollapse` to only use rules that deal with individual stations (that have category numbers from 1 to 69, and thus are below 99), and do not use the rules that involve collapsed cells (which would have numbers of at least 20102 for the collapsed cell Alewife (1) and Brookline (2)). In the first run, those collapsed cells will always be empty ghosts, and they should not be used in defining how the cells be collapsed.

Note also that with the `greedy` option, one would want to specify the zeroes somewhere in the middle of the streak, and possibly across multiple categories of the interacting variable. In our example, specifying `zeroes(36)` would collapse the midday streak of zero counts, but the need to collapse the zeroes for the night day part and on the weekend would still remain, necessitating something like `zeroes(40)` — which, in turn, will likely create overlapping artifacts in the midday section. However, specifying `zeroes(40)` without `zeroes(36)` would take care of all the streaks identified in the output on page 26.

```
. use trip_sample_rules, clear
. wgtcellcollapse collapse, variables(daypart board_id) mincellsize(1) ///
>         zeroes(39 44 49 60) greedy maxcategory(99) ///
>         generate(dpston4) saving(dpston4.do) replace run
  (output omitted)
. wgtcellcollapse collapse, variables(daypart board_id) mincellsize(20) ///
>         strict feed(dpston4) saving(dpston4.do) append run
  (output omitted)
. assert "`r(failed)´" == ""
. wgtcellcollapse collapse, variables(daypart alight_id) mincellsize(1) ///
>         zeroes(2 40 49 50 60) greedy maxcategory(99) ///
>         generate(dpstoff4) saving(dpstoff4.do) replace run
  (output omitted)
. wgtcellcollapse collapse, variables(daypart alight_id) mincellsize(20) ///
>         strict feed(dpstoff4) saving(dpstoff4.do) append run
  (output omitted)
. assert "`r(failed)´" == ""
```

We have finally been able to produce a clean collapse of everything! Note the use of `assert "‘r(failed)’"==""` in the above code snippet to make sure that all cells have the minimal required size of 20.

As a very minor point, there is still some room for improvement in collapsing the cells on the weekend:

```
. tab alight_id dpstoff4 if daypart == 5 & mod(dpstoff4,100*100)>99
```

alight_id	Long ID of the interaction 5084969	5150150	Total
8. Carmenton	0	1	1
11. Dogville	0	5	5
18. East End	0	4	4
24. Framington	0	2	2
26. Grand Junction	0	1	1
30. High Point	0	8	8
36. Irvingtown	0	2	2
44. Limerick	0	2	2
47. Moscow City	0	6	6
50. Ontario Lake	0	1	1
53. Picadilly Square	2	0	2
62. Silver Spring	9	0	9
68. Toledo Town	7	0	7
69. Union Station	123	0	123
Total	141	32	173

Instead of two cells with sizes 141 and 32, it seems like we could produce three cells, with Union Station (69) being its own cell, and everything else split somewhere in the middle.

5.5 The fifth pass of cell collapse and raking: *if* conditions

We will now code the collapsing cells for the weekend at the lowest level allowed by `wgtcellcollapse`, and we will put those custom coded cells upfront before the main run. (Some special treatment had to be given to the zero cells to avoid overlapping cells around Ninth Street (49) for the night and weekend day parts; without the separation, it is getting collapsed in a long overreach all the way up to Redline Circle (60).)

```
. use trip_sample_rules, clear
. wgtcellcollapse collapse, variables(daypart board_id) mincellsize(1) ///
>         zeroes(39 44 49 60) greedy maxcategory(99) ///
>         generate(dpston5) saving(dpston5.do) replace run
  (output omitted)
. wgtcellcollapse collapse, variables(daypart board_id) mincellsize(20) ///
>         strict feed(dpston5) saving(dpston5.do) append run
  (output omitted)
. assert "`r(failed)´" == ""
. wgtcellcollapse collapse, variables(daypart alight_id) mincellsize(1) ///
>         zeroes(2 40 60) greedy maxcategory(99) ///
>         generate(dpstoff5) saving(dpstoff5.do) replace run
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
Processing zero cells...
  Invoking rule 39:40=23940 to collapse zero cells
  replace dpstoff5 = 1023940 if inlist(dpstoff5, 1000039, 1000040)
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 1:2:8=30108 to collapse zero cells
  replace dpstoff5 = 2030108 if inlist(dpstoff5, 2000001, 2000002, 2000008)
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 30:36:39:40:44=53044 to collapse zero cells
  replace dpstoff5 = 2053044 if inlist(dpstoff5, 2000030, 2000036, 2000039, 2000040,
>  2000044)
  (output omitted)
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 53:55:60=35360 to collapse zero cells
  replace dpstoff5 = 5035360 if inlist(dpstoff5, 5000053, 5000055, 5000060)
Pass 0 through the data...
  smallest count = 1 in the cell      1000002
Pass 12 through the data...
  smallest count = 1 in the cell      1000002
  Done collapsing! Exiting...
. wgtcellcollapse collapse if inlist(daypart,4,5) & inrange(alight_id,49,50), ///
>         variables(daypart alight_id) mincellsize(1) ///
>         feed(dpstoff5) zeroes(49) maxcategory(99) saving(dpstoff5.do) append run
Pass 12 through the data...
  smallest count = 1 in the cell      1000002
Processing zero cells...
  Invoking rule 49:50=24950 to collapse zero cells
  replace dpstoff5 = 4024950 if inlist(dpstoff5, 4000049, 4000050)
Pass 12 through the data...
  smallest count = 1 in the cell      1000002
```

```
  Invoking rule 49:50=24950 to collapse zero cells
  replace dpstoff5 = 5024950 if inlist(dpstoff5, 5000049, 5000050)
Pass 12 through the data...
  smallest count = 1 in the cell      1000002
Pass 14 through the data...
  smallest count = 1 in the cell      5024950
  Done collapsing! Exiting...
. * special cells for weekend
. wgtcellcollapse collapse if daypart==5 & inrange(alight_id,1,36), ///
>         variables(daypart alight_id) mincellsize(50) ///
>         strict feed(dpstoff5) saving(dpstoff5.do) append run
Pass 14 through the data...
  smallest count = 1 in the cell      5000026
  Invoking rule 24:26=22426
  replace dpstoff5 = 5022426 if inlist(dpstoff5, 5000024, 5000026)
Pass 15 through the data...
  smallest count = 1 in the cell      5030108
  Invoking rule 11:30108=40111
  replace dpstoff5 = 5040111 if inlist(dpstoff5, 5000011, 5030108)
  (output omitted)
Pass 19 through the data...
  smallest count = 10 in the cell      5043040
  Invoking rule 70126:43040=110140
  replace dpstoff5 = 5110140 if inlist(dpstoff5, 5070126, 5043040)
Pass 20 through the data...
  smallest count = 23 in the cell      5110140
  WARNING: could not find any rules to collapse dpstoff5 == 5110140
Pass 21 through the data...
  smallest count = .i in the cell      1000002
  Done collapsing! Exiting...
. wgtcellcollapse collapse if daypart==5 & inrange(alight_id,44,68), ///
>         variables(daypart alight_id) mincellsize(50) ///
>         strict feed(dpstoff5) saving(dpstoff5.do) append run
Pass 20 through the data...
  smallest count = 1 in the cell      5024950
  Invoking rule 24950:35360=54960
  replace dpstoff5 = 5054960 if inlist(dpstoff5, 5024950, 5035360)
Pass 21 through the data...
  smallest count = 2 in the cell      5000044
  Invoking rule 44:47=24447
  replace dpstoff5 = 5024447 if inlist(dpstoff5, 5000044, 5000047)
  (output omitted)
Pass 25 through the data...
  smallest count = 27 in the cell      5094468
  WARNING: could not find any rules to collapse dpstoff5 == 5094468
Pass 26 through the data...
  smallest count = .i in the cell      1000002
  Done collapsing! Exiting...
. * all other cells
. wgtcellcollapse collapse, variables(daypart alight_id) mincellsize(20) ///
>         strict feed(dpstoff5) saving(dpstoff5.do) append run
Pass 25 through the data...
  smallest count = 1 in the cell      1000002
  Invoking rule 2:8=20208
  replace dpstoff5 = 1020208 if inlist(dpstoff5, 1000002, 1000008)
Pass 26 through the data...
  smallest count = 1 in the cell      2000011
  Invoking rule 11:18=21118
```

```
  replace dpstoff5 = 2021118 if inlist(dpstoff5, 2000011, 2000018)
  (output omitted)
Pass 64 through the data...
  smallest count = 15 in the cell      3054960
  Invoking rule 62:54960=64962
  replace dpstoff5 = 3064962 if inlist(dpstoff5, 3000062, 3054960)
Pass 65 through the data...
  smallest count = 21 in the cell      2200168
  Done collapsing! Exiting...
. assert "`r(failed)´" == ""
```

The special missing value `.i` that appears in the smallest count report is used internally to stop `wgtcellcollapse` after all of the relevant cases selected by the *if* conditions have been processed.

The manual resolution was successful as the following output demonstrates:

```
. tab alight_id dpstoff5 if daypart == 5
```

alight_id	Interactions of daypart alight_id, with some collapsing 5000069	5094468	5110140	Total
8. Carmenton	0	0	1	1
11. Dogville	0	0	5	5
18. East End	0	0	4	4
24. Framington	0	0	2	2
26. Grand Junction	0	0	1	1
30. High Point	0	0	8	8
36. Irvingtown	0	0	2	2
44. Limerick	0	2	0	2
47. Moscow City	0	6	0	6
50. Ontario Lake	0	1	0	1
53. Picadilly Square	0	2	0	2
62. Silver Spring	0	9	0	9
68. Toledo Town	0	7	0	7
69. Union Station	123	0	0	123
Total	123	27	23	173

The resulting do-files can now be applied to producing control totals, and eventually to raking:

```
. use trip_population, clear
. run dpston5.do
. total num_pass , over(dpston5)
Total estimation                  Number of obs   =        719
      1000001: dpston5 = 1000001
      1000002: dpston5 = 1000002
  (output omitted)
      5000011: dpston5 = 5000011
      5026268: dpston5 = 5026268
      5030108: dpston5 = 5030108
      5051836: dpston5 = 5051836
      5093960: dpston5 = 5093960
```

```
---------------------------------------------------------------
        Over |      Total   Std. Err.     [95% Conf. Interval]
-------------+-------------------------------------------------
num_pass     |
     1000001 |       1423   967.7508     -476.9595    3322.959
     1000002 |       7198    4895.91     -2414.011    16810.01
```
(*output omitted*)
```
     5000011 |       1270    834.301      -367.961    2907.961
     5026268 |        557   364.4324     -158.4805    1272.481
     5030108 |        610   263.2061      93.25444    1126.746
     5051836 |        622   215.5712      198.7749    1045.225
     5093960 |        473   261.8954     -41.17225    987.1723
---------------------------------------------------------------

. matrix dpston5 = e(b)

. matrix coleq dpston5 = _one

. matrix rownames dpston5 = dpston5

. run dpstoff5.do

. total num_pass , over(dpstoff5)

Total estimation                  Number of obs   =        719

      1000018: dpstoff5 = 1000018
      1000030: dpstoff5 = 1000030
```
(*output omitted*)
```
      5000069: dpstoff5 = 5000069
      5094468: dpstoff5 = 5094468
      5110140: dpstoff5 = 5110140

---------------------------------------------------------------
        Over |      Total   Std. Err.     [95% Conf. Interval]
-------------+-------------------------------------------------
num_pass     |
     1000018 |        929   360.7303      220.7878    1637.212
     1000030 |       2189   868.0319      484.8161    3893.184
```
(*output omitted*)
```
     5000069 |       2677   895.7917       918.316    4435.684
     5094468 |        432   87.57763      260.0612    603.9388
     5110140 |        423   120.0254      187.3574    658.6426
---------------------------------------------------------------

. matrix dpstoff5 = e(b)

. matrix coleq dpstoff5 = _one

. matrix rownames dpstoff5 = dpstoff5

. use trip_sample_rules, clear

. run dpston5

. run dpstoff5

. gen byte _one = 1

. ipfraking [pw=_one], ctotal(dpston5 dpstoff5) gen(raked_weight5)
 Iteration 1, max rel difference of raked weights = 37.856256
 Iteration 2, max rel difference of raked weights = .06404821
 Iteration 3, max rel difference of raked weights = .00891802
 Iteration 4, max rel difference of raked weights = .00128619
 Iteration 5, max rel difference of raked weights = .00018966
 Iteration 6, max rel difference of raked weights = .00002818
 Iteration 7, max rel difference of raked weights = 4.198e-06
```

```
 Iteration 8, max rel difference of raked weights = 6.257e-07
The worst relative discrepancy of  7.8e-08 is observed for dpstoff5 == 5110140
Target value =       423; achieved value =       423
   Summary of the weight changes
               |   Mean    Std. dev.    Min        Max       CV
---------------+------------------------------------------------
Orig weights   |      1          0        1          1        0
Raked weights  |  26.487      5.754    13.174     38.634   .2172
Adjust factor  | 26.4869              13.1743    38.6339
. whatsdeff raked_weight5
```

Group	Min	Mean	Max	CV	DEFF	N	N eff
Overall	13.17	26.49	38.63	0.2172	1.0472	3654	3489.37

5.6 Informative labels

Once the collapsing rules are finalized, several types of category labels can be attached to the resulting collapsed cells. Using the mechanics of labels in multiple languages (see [R] **label language**), `wgtcellcollapse label` defines three "languages" to describe the cells. The language `numbered_ccells` may be convenient for debugging purposes in fine-tuning the collapsing algorithms, while the language `texted_ccells` would prove useful for `ipfraking_report` in creating human-readable labels. (In Stata SMCL output, the `label language` instructions are clickable, so the user does not have to copy and paste the command, but can click it instead.)

```
. wgtcellcollapse label, var(dpston5)
(language default renamed unlabeled_ccells)
(language numbered_ccells now current language)
(language texted_ccells now current language)
To attach the numeric labels (of the kind "dpston5==1000001"), type:
   label language numbered_ccells
To attach the text labels (of the kind "dpston5==AM Peak; 1. Alewife"), type:
   label language texted_ccells
The original state, which is also the current state, is:
   label language unlabeled_ccells
. wgtcellcollapse label, var(dpstoff5)
To attach the numeric labels (of the kind "dpstoff5==1000018"), type:
   label language numbered_ccells
To attach the text labels (of the kind "dpstoff5==AM Peak; 18. East End"), type:
   label language texted_ccells
The original state, which is also the current state, is:
   label language unlabeled_ccells
. label language numbered_ccells
. tab dpstoff5 if daypart==5
```

Long ID of the interaction	Freq.	Percent	Cum.
daypart==5, alight_id==69	123	71.10	71.10
daypart==5, alight_id==94468	27	15.61	86.71
daypart==5, alight_id==110140	23	13.29	100.00
Total	173	100.00	

```
. label language texted_ccells

. tab dpstoff5 if daypart==5

            Long ID of the interaction |      Freq.     Percent        Cum.
---------------------------------------+-----------------------------------
            Weekend; 69. Union Station |        123       71.10       71.10
Weekend; 44. Limerick to 68. Toledo Tow |         27       15.61       86.71
 Weekend; 1. Alewife to 40. King Street |         23       13.29      100.00
---------------------------------------+-----------------------------------
                                  Total |        173      100.00

. label language unlabeled_ccells

. tab dpstoff5 if daypart==5

Interaction |
       s of |
    daypart |
alight_id, |
  with some |
 collapsing |      Freq.     Percent        Cum.
------------+-----------------------------------
    5000069 |        123       71.10       71.10
    5094468 |         27       15.61       86.71
    5110140 |         23       13.29      100.00
------------+-----------------------------------
      Total |        173      100.00
```

6 Linear calibrated weights

Using the final set of collapsed categories in the simulated transportation data example, let us demonstrate the linear calibration option of `ipfraking`, added since Kolenikov (2014). In mathematical terms, linear weights explicitly solve the minimization problem of finding a set of weights $\{w_{li}, i = 1, \dots, n\}$, where the subscript l stands for *linear* calibration, such that

$$\sum_{i=1}^{n} \frac{(w_{li} - w_{di})^2}{w_{di}} \to \min \tag{6}$$

Deville and Särndal (1992) and Särndal et al. (1992) provide explicit treatment of the problem and the resulting analytical expressions coded in `ipfraking, linear`. The main advantage of linear weight calibration is a much faster computing time. To demonstrate it, we will time the output by using the immediate timing results, `set rmsg on` (see [R] **set**).

```
. set rmsg on
r; t=0.00 14:59:22

. ipfraking [pw=_one], ctotal(dpston5 dpstoff5) nograph gen(raked_weight5)
 Iteration 1, max rel difference of raked weights = 37.856256
 Iteration 2, max rel difference of raked weights = .06404821
 Iteration 3, max rel difference of raked weights = .00891802
 Iteration 4, max rel difference of raked weights = .00128619
 Iteration 5, max rel difference of raked weights = .00018966
 Iteration 6, max rel difference of raked weights = .00002818
 Iteration 7, max rel difference of raked weights = 4.198e-06
```

```
 Iteration 8, max rel difference of raked weights = 6.257e-07
The worst relative discrepancy of  7.8e-08 is observed for dpstoff5 == 5110140
Target value =        423; achieved value =        423
   Summary of the weight changes
              |    Mean    Std. dev.    Min        Max       CV
--------------+--------------------------------------------------
Orig weights  |       1          0         1          1       0
Raked weights |   26.487     5.754    13.174     38.634   .2172
Adjust factor |  26.4869              13.1743    38.6339
r; t=2.16 14:59:24

. ipfraking [pw=_one], ctotal(dpston5 dpstoff5) nograph gen(raked_weight5l) linear
Linear calibration
The worst relative discrepancy of  1.8e-14 is observed for dpstoff5 == 5110140
Target value =        423; achieved value =        423
   Summary of the weight changes
              |    Mean    Std. dev.    Min        Max       CV
--------------+--------------------------------------------------
Orig weights  |       1          0         1          1       0
Raked weights |   26.487    5.7523    12.518     38.204   .2172
Adjust factor |  26.4869              12.5178    38.2040
r; t=0.63 14:59:25

. set rmsg off

. label variable raked_weight5l "Linear calibrated weights"

. compare raked_weight5 raked_weight5l
                                        ---------- difference ----------
                            count       minimum      average     maximum
------------------------------------------------------------------------
raked_w~5<raked_~5l          1896     -1.813144    -.0476911   -3.11e-11
raked_w~5>raked_~5l          1758      2.18e-09     .0514348    2.405758
                       ----------
jointly defined              3654     -1.813144     3.21e-10    2.405758
                       ----------
total                        3654
```

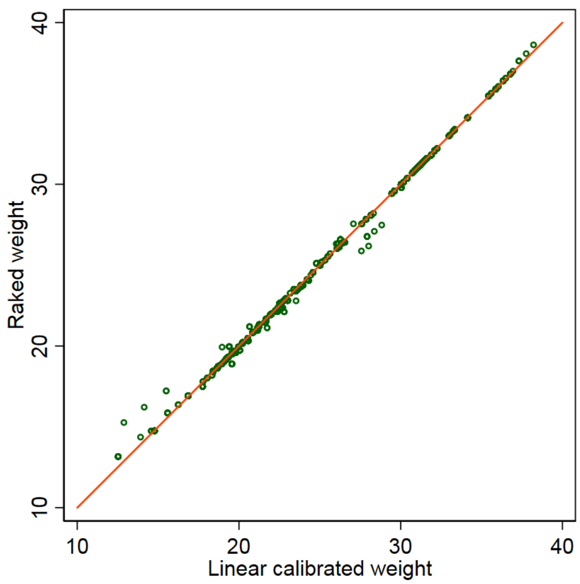

Figure 1: Linear and raked weights

The speed advantages of `linear` calibration are quite clear (0.63 seconds vs. 2.16 seconds), even though raking convergence in 8 iterations is quite fast, in author's experience. It is not unusual to see dozens iterations, and when higher order interactions are being used as raking margins, subtle correlations between the cells arise, slowing down convergence and requiring hundreds of iterations. Linear calibrated and raked weights are very similar to one another, as Figure 1 demonstrates, albeit the lowest of the linearly calibrated weights are slightly smaller than comparable raked weights. As the two methods are distinct, the weights should be expected to agree in general, but the match along the diagonal line of the plot should not be expected to be ideal.

As mentioned before, in the extreme situations, linearly calibrated weights may become negative, which creates additional issues. First, Stata's `svy` commands or estimation commands with `pweight` specifications do not accept negative weights, and produce error messages when such weights are encountered. (This is not a bug, but indeed a welcome behavior.) Second, negative weights are typically difficult to interpret; within a common, although not technically accurate, interpretation of sampling weights as the number of population units that a sampled unit represents, it is puzzling to find a negative number of such population units. The way the author uses the linear calibration functionality of `ipfraking` is to produce "preliminary" sets of weights. If the weights at the low end satisfy the natural range restriction (greater than 0, so as not to produce input data check errors with estimation commands; or sometimes greater than 1, so as to satisfy the "number of population units" interpretation that is often desirable for the clinets), these weights can be "accepted" as final. If they do not, `ipfraking` can be called with trimming syntax such as `trimloabs(1)`. The linear weights can then be used as a starting point to accelerate convergence.

While the general theory of calibrated estimation (Deville and Särndal 1992) ensures that linear calibrated weights (analyzed as Case 1 in that paper) and raked weights (Case 2) are asymptotically equivalent, this equivalence implicitly requires that the scales of the population control matrices are identical. In practice, different control total variables may come from different sources, and some sources may have either different populations to which they can technically be generalized, or come at different scales such as proportions vs. population totals. Nearly every general population dual frame RDD survey that the present author had dealt with would use the American Community Survey data for demographic variables (that would come with the desirable population scaling), and National Health Interview Survey data for phone use variables (cell phone only, landline only, both, or none) that would come in the form of proportions. While the raking version of `ipfraking` would not have any difficulty incorporating both (with the caveat that the final scale of weights will be determined by the *last* variable in the `ctotal()` list), the linear version of weights would try to find a middle point between the population totals that are on the scale of millions, and proportions that are on the scale of about 1. The results would likely be quite strange.

6.1 Other packages with similar functionality

There exist other packages that provide similar basic functionality (i.e., raked weights without trimming). Kolenikov (2014) provided comparisons with `survwgt` (Winter 2002), `ipfweight` (Bergmann 2011), `maxentropy` (Wittenberg 2010), and reported that the weights produced by these packages were identical within numeric accuracy.

Yet another weight calibration package that was published simultaneously with Kolenikov (2014) was `sreweight` (Pacifico 2014). It implements a full range of objective functions from Deville and Särndal (1992), and does so faster than `ipfraking` as the core iterative functionality is implemented in Mata. Finally, Stata 15.1 now provides `svycal` command, undocumented at the time of the writing of this paper, although described and exemplified in detail in Vallian and Dever (2017). Compared to `svycal`, the core functionality of `ipfraking` provides a richer set of trimming specifications. The author compared the weights produced by `ipfraking` with those produced by `sreweight` and `svycal` in the case of basic raking procedure without trimming, and they agree within numeric accuracy:

```
. svycal rake ibn.sex_age ibn.region ibn.race [pw=finalwgt], ///
>         generate(rakedwgt2a) totals(alltotals) nocons
note: 4.region omitted because of collinearity
note: 3.race omitted because of collinearity

. compare rakedwgt2 rakedwgt2a
                                        ---------- difference ----------
                            count       minimum      average     maximum
------------------------------------------------------------------------
rakedwgt2<rakedw~2a          6843     -.0057653    -.0001227   -3.27e-06
rakedwgt2>rakedw~2a          3508      1.56e-07     .0002394    .0038718
                       ----------
jointly defined             10351     -.0057653     1.70e-13    .0038718
                       ----------
total                       10351

. assert reldif(rakedwgt2, rakedwgt2a) < c(epsfloat)
```

Weights produced by `ipfraking` also agree with those produced by R package `survey` (Lumley 2018), namely `survey::calibrate(...,calfun="raking")` function, and those produced by SAS raking macro `RAKE_AND_TRIM()` (Izrael et al. 2017). When trimming options are specified, the results from different packages diverge, as trimming operations can be performed at different stages of raking.

It is somewhat unfortunate that so much effort has gone into replicating the functionality by the different authors. The primary distinction of the current `ipfraking` package is the rich ecosystem that goes along with it, aimed, in its totality, at producing survey weights by a survey organization in a way that is efficient, robust and flexible from coding perspective.

As a practicing survey statistician who needs to experiment with the weights a lot, the present author believes that `ipfraking` is easier to experiment with than `svycal` or `sreweight`, for several reasons. First, `ipfraking` relies on the control totals being carried over from `svy: total` with minimal modifications such as renaming row and column names; passing control totals is more cumbersome with other packages. Second, `ipfraking` produces detailed diagnostics of the problems and oddities it encounters along the way, assisting the survey statistician assess whether the resulting weights are satisfactory. For relatively simpler tasks of producing replicate weights and calibrating them at the same time, `survwgt` provides a much easier syntax. Coding the task with `ipfraking` or any other package would require explicit cycles.

From code development perspective, the present author believes that relying on matching the order of control totals and variables, as required by all other user-contributed packages, creates a potential for errors that are easy to make and difficult to catch. With either `ipfraking` and the official `svycal`, the risk that a control total figure would be associated with a wrong category of the control total variable is much lower, as they pair the values and the categories in a single object (via the names attached to the control total matrices) or a single syntax component `value.variable =` # specification of `svycal`. (The matrix naming is different between `ipfraking` and `svycal`, though, so a conversion tool, `totalmatrices`, is provided in this update.)

Additionally, `ipfraking` can incorporate variables that sum up to different totals, e.g., totals from different sources or from different years, or totals and proportions, in case unified data are not available, with the side effect of producing weights whose totals agree with the *last* control total variable. Without trimming, doing so ensures that *proportions* for each calibration variables are satisfied. As `maxentropy`, `sreweight` and `svycal` produce weights by optimization, with the goal of satisfying all totals simultaneously, it is unclear what the properties of the resulting weight would be when the control totals differ between variables, and whether the resulting weights would produce marginal proportions that agree between the control totals and calibrated weights.

Compared to `ipfraking`, the official `svycal` command handles interactions far more graciously, and consistently with the Stata user experience of using factor variables in regression models. It creates the necessary interactions internally on the fly, while `ipfraking` requires explicit creating of interaction variables.

In the end of the day, the choice of the package is a matter of personal preference, package familiarity, and coding style.

Acknowledgements

The author is grateful to the anonymous referee for a thorough review and thoughtful suggestions, to Tom Guterbock for bug reports and functionality suggestions, and to Jason Brinkley for extensive comments and critique. The opinions stated in this paper are of the author only, and do not represent the position of Abt Associates.

7 References

AAPOR. 2014. *AAPOR Terms and Conditions for Transparency Certification.* The American Association for Public Opinion Research. Available at http://www.aapor.org/AAPOR_Main/media/MainSiteFiles/TI-Terms-and-Conditions-10-4-17.pdf.

Andridge, R. R., and R. J. Little. 2010. A Review of Hot Deck Imputation for Survey Non-response. *International statistical review* 78(1): 40–64.

Bergmann, M. 2011. IPFWEIGHT: Stata module to create adjustment weights for surveys. Statistical Software Components, Boston College Department of Economics. RePEc:boc:bocode:s457353.

Binder, D. A., and G. R. Roberts. 2003. Design-based and Model-based Methods for Estimating Model Parameters. In *Analysis of Survey Data*, ed. R. L. Chambers and C. J. Skinner, chap. 3. New York: John Wiley & Sons.

Deville, J. C., and C. E. Särndal. 1992. Calibration Estimators in Survey Sampling. *Journal of the American Statistical Association* 87(418): 376–382.

Deville, J. C., C. E. Särndal, and O. Sautory. 1993. Generalized Raking Procedures in Survey Sampling. *Journal of the American Statistical Association* 88(423): 1013–1020.

Gould, W. 2003. Stata tip 3: How to be assertive. *Stata Journal* 3(4).

Groves, R. M., D. A. Dillman, J. L. Eltinge, and R. J. A. Little. 2001. *Survey Nonresponse*. Wiley Series in Survey Methodology, Wiley-Interscience.

Holt, D., and T. M. F. Smith. 1979. Post Stratification. *Journal of the Royal Statistical Society, Series A* 142(1): 33–46.

Horvitz, D. G., and D. J. Thompson. 1952. A Generalization of Sampling Without Replacement From a Finite Universe. *Journal of the American Statistical Association* 47(260): 663–685.

Izrael, D., M. P. Battaglia, A. A. Battaglia, and S. W. Ball. 2017. You Do Not Have To Step On The Same Rake: SAS Raking Macro—Generation IV. In *SAS Global Forum*. Available at https://support.sas.com/resources/papers/proceedings17/0470-2017-poster.pdf.

Kalton, G. 2009. Methods for oversampling rare subpopulations in social surveys. *Survey Methodology* 35(2): 125–141. Https://www150.statcan.gc.ca/n1/en/pub/12-001-x/2009002/article/11036-eng.pdf?st=HXLvKAiS.

Kalton, G., and D. Anderson. 1986. Sampling rare populations. *Journal of the Royal Statistical Society. Series A (General)* 149(1): 65–82.

Kolenikov, S. 2010. Resampling inference with complex survey data. *The Stata Journal* 10: 165–199.

———. 2014. Calibrating survey data using iterative proportional fitting. *The Stata Journal* 14(1): 22–59.

———. 2016. Post-stratification or non-response adjustment? *Survey Practice* 9(3). Available at http://www.surveypractice.org/index.php/SurveyPractice/article/view/315.

Kolenikov, S., and H. Hammer. 2015. Simultaneous Raking of Survey Weights at Multiple Levels. *Survey Methods: Insights from the Field* Special issue on Weighting: Practical Issues and How to Approach. Retrieved from https://surveyinsights.org/?p=5099.

Korn, E. L., and B. I. Graubard. 1995. Analysis of Large Health Surveys: Accounting for the Sampling Design. *Journal of the Royal Statistical Society, Series A* 158(2): 263–295.

———. 1999. *Analysis of Health Surveys*. John Wiley and Sons.

Kott, P. S. 2006. Using Calibration Weighting to Adjust for Nonresponse and Coverage Errors. *Survey Methodology* 32(2): 133–142.

———. 2009. Calibration Weighting: Combining Probability Samples and Linear Prediction Models. In *Sample Surveys: Inference and Analysis*, ed. D. Pfeffermann and C. R. Rao, vol. 29B of *Handbook of Statistics*, chap. 25. Oxford, UK: Elsevier.

Lumley, T. S. 2018. Package ‘survey’, v. 3.34.

Pacifico, D. 2014. sreweight: A Stata command to reweight survey data to external totals. *The Stata Journal* 14(1): 4–21.

Park, D. K., A. Gelman, and J. Bafumi. 2004. Bayesian Multilevel Estimation with Poststratification: State-Level Estimates from National Polls. *Political Analysis* 12(4): 375–385.

Pew Research Center. 2012. Assessing the Representativeness of Public Opinion Surveys. Technical report, Pew Research Center for People and Press. Available at http://www.people-press.org/files/legacy-pdf/Assessing the Representativeness of Public Opinion Surveys.pdf.

Pfeffermann, D. 1993. The role of sampling weights when modeling survey data. *International Statistical Review* 61: 317–337.

Rubin, D. B. 1976. Inference and missing data. *Biometrika* 63(3): 581–592.

Särndal, C.-E. 2007. The calibration approach in survey theory and practice. *Survey Methodology* 33(2): 99–119.

Särndal, C.-E., B. Swensson, and J. Wretman. 1992. *Model Assisted Survey Sampling*. New York: Springer.

Thompson, M. E. 1997. *Theory of Sample Surveys*, vol. 74 of *Monographs on Statistics and Applied Probability*. New York: Chapman & Hall/CRC.

Vallian, R., and J. Dever. 2017. *Survey Weights: A Step-by-step Guide to Calculation*. College Station, TX: Stata Press.

Winter, N. 2002. SURVWGT: Stata module to create and manipulate survey weights. Statistical Software Components, Boston College Department of Economics. RePEc:boc:bocode:s427503.

Wittenberg, M. 2010. An introduction to maximum entropy and minimum cross-entropy estimation using Stata. *Stata Journal* 10(3): 315–330.

About the author

Stanislav (Stas) Kolenikov is Principal Scientist at Abt Associates. His work involves applications of statistical methods in data collection for public opinion research, public health, transportation, and other disciplines that utilize collection of survey data. Within survey methodology, his expertise includes advanced sampling techniques, survey weighting, calibration, missing data imputation, variance estimation, nonresponse analysis and adjustment, small area estimation, and mode effects. Besides survey statistics, Stas has extensive experience developing and applying statistical methods in social sciences, with focus on structural equation modeling and microeconometrics. He has been writing Stata programs since 1998 when Stata was version 5.