On the non-recursive implementation of multistage sampling without replacement

Philippe Aubry

OFB - Office français de la biodiversité - Direction surveillance, évaluation, données - Unité données et appui méthodologique, Saint Benoist, BP 20, F-78612 Le Perray-en-Yvelines, France

A B S T R A C T

Variance estimation in multistage sampling without replacement usually requires considerable computational effort. One option is to implement explicit formulas on a computer, at least for some specific sampling designs. This approach becomes quite cumbersome to handle beyond two stages, both from the formulation and computer implementation points of view. Another option is to provide a general method to compute variance estimates for any number of stages. Such an approach may involve data structures and estimators which are recursively defined.

- The solution we present in this article is intended to be both general and computationally efficient by relying on a full-iterative implementation.
- The definition of the estimators remains implicit as in the recursive approach, but is expressed in terms of recurrence relations translated into iterative algorithms.
- These algorithms rely only on (dense) array data structures. Moreover, most of the necessary computer memory is only used during preliminary steps and is not required when performing the statistical calculations.

© 2021 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/)

A R T I C L E   I N F O
Method name: Iterative tree-based computation in multistage sampling without replacement
Keywords: Multistage sampling, Horvitz-Thompson estimator, Expansion estimator, Sen-Yates-Grundy variance, Approximate variance estimator, Self-weighted design, Array data structures, Non-recursive algorithms
Article history: Received 1 July 2021; Accepted 16 October 2021; Available online 20 October 2021

E-mail address: philippe.aubry@ofb.gouv.fr

https://doi.org/10.1016/j.mex.2021.101553
2215-0161/© 2021 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/)
Specifications Table

| Subject Area: | Computational statistics |
|---------------|--------------------------|
| More specific subject area: | Complex sampling designs; Variance estimation techniques |
| Method name: | Iterative tree-based computation in multistage sampling without replacement |
| Name and reference of original method: | D. Bellhouse, Computation of variance-covariance estimates for general multistage sampling designs, in: M. M. Barritt, D. Wishart (Eds.), COMPSTAT 1980: proceedings in computational statistics. 4th symposium held at Edinburgh 1980, Physica-Verlag, Vienna, Austria, 1980, pp. 57–63. |
| | Bellhouse, Computing methods for variance estimation in complex surveys, J. Off. Stat. (1985) 323–329. |
| | D. Rylett, D. Bellhouse, TREES: a computer program for complex surveys, in: Proceedings of the survey research methods section, ASA, session XIX, sampling error: methodology, software and applications, American Statistical Association, Alexandria, VA, 1988, pp. 694–697. |
| Resource availability: | not applicable |

Method details

In multistage sampling without replacement, a general method for unbiased variance estimation for the expansion estimator (i.e. the extension of the Horvitz-Thompson estimator) suitable for computer implementation was first proposed by Durbin [1], improved by Raj [2] and extended by Rao [3]. For instance, with two-stage sampling, this method consists in estimating the first- and second-stage variance components separately and combining them to form an unbiased estimator of overall sampling variance (e.g. [4, sec. 4.3.2] or [5, sec. 7.2.2]). It follows that both the expansion estimator and the sampling variance estimator may be defined recursively which means that the estimator symbol appears on both sides of its definition (e.g. [5, p. 158]).

Generally speaking, there is a dichotomy between the recursive definition of a calculation, which is implicit, simple and quick to implement on a computer, and its iterative counterpart, usually more complicated to work out. Owing to the recursive definition of the hierarchical nesting of sampling units and of the sampling variance, it follows that the simplest and most direct way to implement multistage sampling on a computer is to rely on a full recursive implementation, based on a recursively defined data structure and on recursive programming (for recursive programming the reader is referred to Rohlf [6] or Rubio-Sánchez [7]).

In imperative programming languages such as FORTRAN, Pascal, C/C++ and so on, the appropriate recursive data structure (a tree) is constructed using dynamically allocated pointers. Recursion is supported by allowing a function (or procedure) to call itself. In this case, recursive calls are automatically managed at the call stack level. For turning a recursive function (or procedure) into an iterative one, one approach is simulating recursion using a manual stack in lieu of relying on the call stack. By using pointers for tree representation and a manual stack for tree traversal we obtain a mixed implementation, that is, recursive for the data structure but iterative for tree traversal (by simulating the recursion). This is precisely what has been proposed by Bellhouse [8,9] and Rylett and Bellhouse [10].

Multistage sampling is also implemented “recursively” in softwares used by some statistical agencies. This is the case for instance of POULPE, a SAS macro program in use at INSEE in France [11,12]. We guess that the recursion in POULPE is simulated, in a way that has not, however, been detailed in the literature. The main R packages devoted to probability sampling are ‘sampling’ and ‘survey’ [13]. In the ‘sampling’ package [14], to our knowledge variance estimation for multistage sampling is not handled. Conversely, package ‘survey’ [15,16] estimates the sampling variance by relying on recursive programming (the function ‘multistage’ calls itself). As far as we know this is also the case for the ‘ReGenesees’ package which inherits from ‘survey’ regarding multistage sampling [17].

In this technical article we show how multistage sampling can be implemented in a full iterative way, that is without simulating the recursion nor using pointers, by relying on simple (flat) array data structures. Moreover, the solution we propose does not require maintaining in memory all the tree information for performing the statistical computations.
The full-iterative computer implementation detailed here should in particular be useful for handling very large multistage sampling configurations and also for Monte Carlo simulations. It should also be of some pedagogical interest for the in-depth teaching of multistage sampling for a graduate course in computational statistics (even if we have to recognize that this is a somewhat specific technical topic).

Hence, for multistage sampling without replacement, the aim of the present article is to provide (1) a complete recurrent formulation thanks to an appropriate notation and (2) a comprehensive set of algorithms for full-iterative implementation, ensuring efficient statistical computations. To our knowledge, these two technical points have not to date been documented in the statistical computing literature. We focus here on total estimation based on the Horvitz–Thompson estimator, and fixed size designs at all stages. Other estimators can be used without questioning the general organization of the algorithms, and variable size designs can be accommodated in the general case by referring to the Horvitz–Thompson variance instead of the Sen-Yates-Grundy variance (see for instance [18, sec. 7.3] or [4, sec. 4.3.2]).

**Recurrence formulas**

In this section, we introduce basic notation conventions that we think appropriate to write recurrence formulas for multistage sampling. Then we express multistage sampling and inclusion probabilities with our notations, and we provide the formulas for estimation: (i) in the general case, that is possibly with unequal inclusion probabilities for all designs, at all stages; (ii) for the particular case of equal probabilities for all designs, at all stages; and finally (iii) for self-weighted sampling.

**Notations**

Throughout the formulas section, indices will be noted by a lower case letter, the set of which will be noted by the same letter, but in upper case. For instance, we will have the set of indices \( I = \{ i \}_{i=1}^{N} \) and we will be able to describe the elements indicated by \( i \) simply by using set membership, that is \( i \in I \). For the sake of notation simplicity, in what follows we will use \( i < j \in I \) in lieu of \( i < j, i, j \in I \).

At each level \( 0 \leq \ell \leq L \) of the tree corresponding to the nested structure of multistage sampling units, the population is denoted \( \mathcal{U}_{\ell} = \{ u_{\ell,i} \}_{i \in I_{\ell}} \) where \( I_{\ell} = \{ 1, \ldots, N_{\ell} \} \) is a set of indices (i.e. the units labels) as in Daffin et al. [19, p. 512]. The root is at level \( \ell = 0 \) in the tree, and will be denoted as the singleton set \( \mathcal{U}_{0} = \{ u_{0,1} \} \). At levels \( 0 \leq \ell < L \), sampling units are populations of units from level \( \ell + 1 \). At last level \( \ell = L \) the sampling units are elementary units (that is units which are not further decomposed) which constitute a population of size \( N_{L} = N \). Considering clusters of elementary units as ultimate sampling units is merely introducing a level with 100% sampling fraction within the selected clusters. We will denote in a recurrent way:

\[
\begin{align*}
\text{for } \ell = 0 & \quad \mathcal{U}_{0} = \{ u_{0,1} \} & N_{0} = 1 & \quad I_{0} = \{ 1 \} \\
\text{for } \ell = 1, \ldots, L & \quad \mathcal{U}_{\ell} = \bigcup_{i \in I_{\ell-1}} \mathcal{U}_{\ell,i} & N_{\ell} = \sum_{i=1}^{N_{\ell-1}} N_{\ell,i} & \quad I_{\ell} = \{ 1, 2, \ldots, N_{\ell} \}
\end{align*}
\]

with \( \mathcal{U}_{\ell,i} \) the set of sampling units at level \( \ell \) which constitute the population corresponding to the parent unit \( u_{\ell-1,i} \) and \( N_{\ell,i} \) the cardinality of \( \mathcal{U}_{\ell,i} \). The index sequence \( I_{\ell} \) is partitioned into sub-sequences corresponding to the \( N_{\ell-1} \) parent units, that is:

\[
I_{\ell} = (a_{1}, \ldots, b_{1}, a_{2}, \ldots, b_{2}, \ldots, a_{t_{1}}, \ldots, b_{t_{1}}, \ldots, a_{t_{2}}, \ldots, b_{t_{2}}, \ldots, a_{t_{L}}, \ldots, b_{t_{L}})
\]

with \( a_{1} = 1, a_{t_{\ell}} = b_{\ell} + 1 \) and \( b_{i} = \sum_{j=1}^{i} N_{\ell,j} \) for \( 1 \leq i < N_{\ell-1} \), and finally \( b_{N_{\ell-1}} = N_{\ell} \).

Hence we have a sequence of index sub-sequences \( (I_{t_{1}}, I_{t_{2}}, \ldots, I_{t_{L}}, I_{t_{L-1}}, \ldots, I_{t_{1}}) \) which corresponds to the sequence of unit labels for sub-populations \( (\mathcal{U}_{t_{1}}, \mathcal{U}_{t_{2}}, \ldots, \mathcal{U}_{t_{L}}, \ldots, \mathcal{U}_{t_{L-1}}) \). For better comprehension, our notations are illustrated in Fig. 1 (for \( L = 2 \)).
Multistage sampling design

At stage $\ell = 0$ there is no sampling since by definition the unit $u_{0,1}$ has the probability $\pi_{0,1} = 1$ to be included so $S_0 = \{1\}$. At stages $1 \leq \ell \leq L$ one can apply a sampling design $d_{\ell,i}$ to each label sub-sequence $l_{\ell,i}$ associated with the units of the sub-population $U_{\ell,i}$. This sampling will be denoted by referring to the function $s_{\ell,i} = d_{\ell,i}(l_{\ell,i})$ with $s_{\ell,i}$ a sample of size $n_{\ell,i}$ ($1 < n_{\ell,i} \leq N_{\ell,i}$). The multistage sampling design may be written recurrently as:

$$
\begin{align*}
\text{for } \ell = 0 & \quad S_0 = \{1\} \\
\text{for } \ell = 1, \ldots, L \text{ and } i \in S_{\ell-1} & \quad s_{\ell,i} = d_{\ell,i}(l_{\ell,i}) \quad s_{\ell} = \bigcup_{i \in S_{\ell-1}} s_{\ell,i} 
\end{align*}
$$

(1)

Once the $L$-stage sampling design is applied, we obtain a sample $S_L = S$ including $n_L = n$ elementary units for gathering values of the variable of interest $y$.

The recurrent formulation of multistage sampling allows any design $d_{\ell,i}$ to be used. Theoretically, the definition of designs at level $\ell > 1$ may depend on the sample $S_{\ell-1}$. Moreover, samplings at level $\ell > 1$ among labels $l_{\ell,i}$ and $l_{\ell,j}$ for $i \neq j \in S_{\ell-1}$ are not necessarily independent. From this generality it follows that this class of sampling designs may be rather complex. In practice, only a sub-class of multistage sampling designs is used, by imposing both invariance and independence. The invariance property implies that the same design $d_{\ell,i}$ will be applied to $l_{\ell,j}$ whatever the sample $s_{\ell-1}$ which includes the parent unit $u_{\ell-1,i}$. The independence property implies that sampling inside each set of labels belonging to the same level $\ell$ will be applied independently. For more details about invariance and independence, the reader is referred to Särndal et al. [4].

Inclusion probabilities

With each design $d_{\ell,i}$ is associated a vector $\pi_{\ell,i}$ containing $N_{\ell,i}$ first-order inclusion probabilities:

$$
\pi_{\ell,i} = (\pi_{\ell,a_1}, \ldots, \pi_{\ell,j}, \ldots, \pi_{\ell,b_1})
$$

with

$$
\sum_{j=a_i}^{b_i} \pi_{\ell,j} = n_{\ell,i}
$$
and a matrix $\mathbf{\Pi}_{\ell,i}$ containing $N_{\ell,i} \times N_{\ell,i}$ second-order (or joint) inclusion probabilities:

$$
\begin{pmatrix}
\pi_{\ell,a_{ij}} & \cdots & \pi_{\ell,a_{ib}} \\
\vdots & \ddots & \vdots \\
\pi_{\ell,j_{a_{ij}}} & \cdots & \pi_{\ell,j_{ib}} \\
\vdots & \ddots & \vdots \\
\pi_{\ell,b_{a_{ij}}} & \cdots & \pi_{\ell,b_{ib}}
\end{pmatrix}
$$

with

$$
\sum_{j=0}^{b_1} \sum_{k=0}^{b_j} \pi_{\ell,jk} = n_{\ell,i}^2
$$

For $U_\ell$, we may denote first-order inclusion probabilities as the sequence of the probability vectors for all sampling designs at level $\ell$, and the resulting vector is indexed by indices from $I_\ell$, that is:

$$
\mathbf{\pi}_\ell = \left( \mathbf{\pi}_{\ell,1}, \mathbf{\pi}_{\ell,2}, \ldots, \mathbf{\pi}_{\ell,l}, \ldots, \mathbf{\pi}_{\ell,N_{\ell,i}} \right)
$$

At level $\ell$, we denote $\pi_{\ell,i}^*$ the overall (or final) inclusion probability for unit $u_{\ell,i}$ resulting from the application of sampling designs from level 1 to $\ell$. As a result of both invariance and independence for the sampling designs, we get the recurrence:

$$
\begin{align*}
\text{for } & \ell = 0 \text{ and } j = 1 \quad \pi_{\ell,i}^* = 1 \\
\text{for } & \ell = 1, \ldots, L; i \in l_{\ell-1} \text{ and } j \in l_{\ell,i} \quad \pi_{\ell,i}^* = \pi_{\ell,j} \pi_{\ell-1,i}^*
\end{align*}
$$

(2)

In the literature devoted to multistage sampling, this relation is sometimes known as the selection equation [20,21].

**Unequal probability sampling designs at all stages**

In this section we consider the recurrence formulation for the general case, allowing unequal inclusions probabilities at all stages, for all sampling designs.

The total may be estimated using an expansion estimator according to the following recurrence:

$$
\begin{align*}
\text{for } & \ell = L \text{ and } i \in s_\ell \quad \hat{Y}_{\ell,i} = y_i \\
\text{for } & \ell = L, \ldots, 1 \text{ and } i \in s_{\ell-1} \quad \hat{Y}_{\ell-1,i} = \sum_{j \in s_{\ell,i}} \frac{\hat{Y}_{\ell,j}}{\pi_{\ell,j}}
\end{align*}
$$

(3)

The total estimator is then $\hat{Y}_{0,1}$. Its sampling variance may be written using the recurrence:

$$
\begin{align*}
\text{for } & \ell = L \text{ and } i \in l_\ell \quad V_{\ell,i} = 0 \\
\text{for } & \ell = L, \ldots, 1 \text{ and } i \in l_{\ell-1} \quad V_{\ell-1,i} = V_{\ell-1,i}^* + \sum_{j \in l_{\ell,i}} \frac{V_{\ell,j}}{\pi_{\ell,j}}
\end{align*}
$$

(4)

The sampling variance is then $V_{0,1}$. When joint inclusion probabilities can be managed (that is, both calculated and stored), and because we consider only fixed-size designs, the term $V_{\ell-1,i}^*$ may be the Sen-Yates-Grundy variance:

$$
V_{\ell-1,i}^* = \sum_{j \in l_{\ell,j}} \sum_{k \in l_{\ell,k}} (\pi_{\ell,j} \pi_{\ell,k} - \pi_{\ell,jk}) \left( \frac{Y_{\ell,j}}{\pi_{\ell,j}} - \frac{Y_{\ell,k}}{\pi_{\ell,k}} \right)^2
$$

(5)
The sampling variance estimator may be written using the recurrence:

for $\ell = L$ and $i \in s_\ell$ \hspace{1cm} $\hat{V}_{\ell,i} = 0$

for $\ell = L, \ldots, 1$ and $i \in s_{\ell-1}$ \hspace{1cm} $\hat{V}_{\ell-1,i} = \hat{V}_{\ell-1,i}^* + \sum_{j \in s_{\ell,i}} \frac{\hat{V}_{\ell,i}}{\pi_{\ell,j}}$ \hspace{1cm} (6)

The sampling variance estimator is then $\hat{V}_{0,1}$. The term $\hat{V}_{\ell-1,i}^*$ may be the Sen-Yates-Grundy unbiased variance estimator:

$$\hat{V}_{\ell-1,i}^* = \sum_{j < k \in s_{\ell,i}} \left( \frac{\pi_{\ell,j} \pi_{\ell,k} - \pi_{\ell,jk}}{\pi_{\ell,jk}} \right) \left( \frac{\hat{Y}_{\ell,j}}{\pi_{\ell,j}} - \hat{Y}_{\ell,k} \right)^2$$ \hspace{1cm} (7)

with $\pi_{\ell,jk} > 0$. At least for high entropy designs, when joint inclusion probabilities cannot be managed, a (biased) variance estimator may be used instead of (7) (see the references cited by Aubry [22]). One possibility is to write [5,23]:

$$\hat{V}_{\ell-1,i} = \sum_{j \in s_{\ell,i}} \frac{c_{\ell,j}}{\pi_{\ell,j}} \left( \hat{Y}_{\ell,j} - \hat{V}_{\ell,j}^* \right)^2$$ \hspace{1cm} (8)

with

$$\hat{V}_{\ell,j}^* = \pi_{\ell,j} \frac{\sum_{m \in s_{\ell,i}} \hat{V}_{\ell,m}}{\sum_{m \in s_{\ell,i}} c_{\ell,m}}$$ \hspace{1cm} (9)

and $c_{\ell,j} = (1 - \pi_{\ell,j})n_{\ell,i}/(n_{\ell,i} - 1)$ for $j \in s_{\ell,i}$.

When all sampling designs are PPSWOR (Probability Proportional to Size Without Replacement), then the size is computed at all levels $1 \leq \ell < L$ from the values defined for the population of elementary units. For $i \in I_\ell$ and $\ell = L$, we denote $x_i$ the value for the size-variable, associated with the elementary unit $u_{\ell,i}$. The total $\sum x_i$, can be decomposed by the following recurrence:

for $\ell = L$ and $i \in I_\ell$ \hspace{1cm} $X_{\ell,i} = x_i$

for $\ell = L, \ldots, 1$ and $i \in I_{\ell-1}$ \hspace{1cm} $X_{\ell-1,i} = \sum_{j \in I_{\ell,i}} X_{\ell,j}$ \hspace{1cm} (10)

The total is then $X_{0,1}$.

Equal probability sampling designs at all stages

When every population $U_{\ell,i}$ is sampled using SRSWOR (Simple Random Sampling Without Replacement) of size $n_{\ell,i}$ out of $N_{\ell,i}$ ($1 < n_{\ell,i} \leq N_{\ell,i}$), then formulas become considerably simpler. Indeed, up to second order, inclusion probabilities are simply (for $j \neq k \in I_{\ell,i}$):

$$\pi_{\ell,j} = \frac{n_{\ell,i}}{N_{\ell,i}}$$

$$\pi_{\ell,jk} = \frac{n_{\ell,i}(n_{\ell,i}-1)}{N_{\ell,i}(N_{\ell,i}-1)}$$ \hspace{1cm} (11)

It follows that the recurrence for the total estimator is now written:

for $\ell = L$ and $i \in s_\ell$ \hspace{1cm} $\hat{V}_{\ell,i} = y_i$

for $\ell = L, \ldots, 1$ and $i \in s_{\ell-1}$ \hspace{1cm} $\hat{V}_{\ell-1,i} = \frac{N_{\ell,i}}{n_{\ell,i}} \sum_{j \in s_{\ell,i}} \hat{V}_{\ell,j}$ \hspace{1cm} (12)
The recurrence for sampling variance is now:

\[
\begin{align*}
&\text{for } \ell = L \text{ and } i \in I_{\ell} \quad V_{\ell,i} = 0 \\
&\text{for } \ell = L, \ldots, 1 \text{ and } i \in I_{\ell-1} \quad V_{\ell-1,i} = N_{\ell,i}^2 \left(1 - \frac{n_{\ell,i}}{N_{\ell,i}}\right) \frac{S_{\ell,i}^2}{n_{\ell,i}} + \frac{N_{\ell,i}}{n_{\ell,i}} \sum_{j \in I_{\ell,i}} V_{\ell,j}
\end{align*}
\]

with

\[
S_{\ell,i}^2 = \frac{1}{N_{\ell,i} - 1} \sum_{j \in I_{\ell,i}} \left(Y_{\ell,j} - \bar{Y}_{\ell,i}\right)^2
\]

and

\[
\bar{Y}_{\ell,i} = \frac{1}{N_{\ell,i}} \sum_{j \in I_{\ell,i}} Y_{\ell,j}
\]

and accordingly, the recurrence for the unbiased sampling variance estimator is:

\[
\begin{align*}
&\text{for } \ell = L \text{ and } i \in S_{\ell} \quad \hat{V}_{\ell,i} = 0 \\
&\text{for } \ell = L, \ldots, 1 \text{ and } i \in S_{\ell-1} \quad \hat{V}_{\ell-1,i} = N_{\ell,i}^2 \left(1 - \frac{n_{\ell,i}}{N_{\ell,i}}\right) \hat{S}_{\ell,i}^2 + \frac{N_{\ell,i}}{n_{\ell,i}} \sum_{j \in S_{\ell,i}} \hat{V}_{\ell,j}
\end{align*}
\]

with

\[
\hat{S}_{\ell,i}^2 = \frac{1}{n_{\ell,i} - 1} \sum_{j \in S_{\ell,i}} \left(Y_{\ell,j} - \hat{\bar{Y}}_{\ell,i}\right)^2
\]

and

\[
\hat{\bar{Y}}_{\ell,i} = \frac{1}{n_{\ell,i}} \sum_{j \in S_{\ell,i}} Y_{\ell,j}
\]

Self-weighted multistage sampling

With our notations, the weight of an elementary unit \(u_{\ell,i}\) is \(w_{\ell,i} = 1/\pi_{\ell,i}^*\) for \(i \in I_{\ell}\) and \(\ell = L\). A design is self-weighted when all elementary units have the same weight that is, for \(\ell = L\), when \(\pi_{\ell,i}^* = \pi_{\ell,j}^*\) (\(i \neq j \in I_{\ell}\)). For the sake of simplicity, this common inclusion probability will be denoted \(\pi^*\).

Let \(X_{\ell,i}\) be the size of the sampling unit \(u_{\ell,i}\) in number of underlying elementary units. We have the following definition by recurrence:

\[
\begin{align*}
&\text{for } \ell = L \text{ and } i \in I_{\ell} \quad X_{\ell,i} = 1 \\
&\text{for } \ell = L, \ldots, 1 \text{ and } i \in I_{\ell-1} \quad X_{\ell-1,i} = \sum_{j \in I_{\ell,i}} X_{\ell,j}
\end{align*}
\]

Obtaining a self-weighted \(L\)-stage sampling design requires the selection of samples of constant size \(m_{\ell}\) among units \(U_{\ell,i} (i \in I_{\ell-1})\) for \(\ell = 1, \ldots, L\) with a probability proportional to their size, that is:

\[
\text{for } \ell = 1, \ldots, L; \ i \in I_{\ell-1} \text{ and } j \in I_{\ell,i} \quad \pi_{\ell,j} = m_{\ell} X_{\ell,j} \sum_{k \in I_{\ell,j}} X_{\ell,k}
\]

Applying recurrence (2), it follows that the constant overall inclusion probability for elementary units is equal to \(\pi^* = n/N\) with:

\[
n = \prod_{\ell=1}^{L} m_{\ell}
\]

At last stage \(L\), expression (16) shows that it is necessary to include elementary units with the following probability:

\[
\text{for } \ell = L; \ i \in I_{\ell-1} \text{ and } j \in I_{\ell,i} \quad \pi_{\ell,j} = \frac{m_{\ell}}{N_{\ell,i}}
\]

which does not depend on \(j\). In the most general case, for \(\ell = 1, \ldots, L\), cardinalities \(N_{\ell,i}\) (\(i \in I_{\ell-1}\)) differ, and therefore sizes \(X_{\ell,j}\) (\(j \in I_{\ell,i}\)) also differ. In this case, units at level \(1 \leq \ell < L\) need to be selected using PPSWOR with inclusion probabilities (16) whilst elementary units have to be selected using SRSWOR with probabilities (18). Although the last stage is SRSWOR we rely here on the general formulation.
In the simplest case, for \( \ell = 1, \ldots, L \) cardinalities are equal, that is, \( M_\ell = N_{i, i} (i \in I_{\ell-1}) \), and therefore sizes \( X_{i,j} = N/N_\ell \) \((j \in I_{\ell,i})\) are also equal and the same arises for first-order inclusion probabilities \( \pi_{i,j} = m_\ell/M_\ell \) \((j \in I_{\ell,i})\) with \( 1 < m_\ell \leq M_\ell \). Applying recurrence (2) it follows, as expected:

\[
\pi^* = \frac{\prod_{\ell=1}^{L} m_\ell}{\prod_{\ell=1}^{L} M_\ell} = \frac{n}{N}
\]

(19)

In this limiting case, sampling units are selected by SRSWOR at each stage with probabilities \( \pi_{i,j} = m_\ell/M_\ell \).

**Algorithms**

In this section we provide algorithms for full-iterative computer implementation of multistage sampling. The principle that guided algorithm elaboration is that the tree corresponding to the hierarchical nesting of sampling units is represented by arrays of which only a part remains useful for statistical computations. Hence, the major part of the required computer memory is used only temporarily and can be released as soon as the preliminary steps are achieved. Moreover, no sparse arrays are used so that no memory places are wasted.

Although we do not provide a theoretical analysis of time and space complexity of the proposed algorithms — a topic far beyond the scope of this paper — we are confident in their efficiency, especially when implemented in a compiled programming language usually used for statistical computing, such as C/C++ or FORTRAN (possibly for inter-operating with the R programming environment).

**Notations**

Algorithms (procedures or functions) are given as pseudocode, with classical control structures. There are three possible passing modes for a procedure parameter depending on whether it is only on input ("in" mode; its value remains unchanged), only on output ("out" mode; its value is defined within the procedure) or both on input/output ("in-out" mode; its value is modified within the procedure). We put a down-arrow, an up-arrow or a down-up-arrow above a parameter's symbol for specifying in, out, and in-out passing modes, respectively. By definition, all parameters of a function are in input-only passing mode and therefore down-arrows will be omitted.

**Hierarchical nesting of units**

The hierarchical nesting of units leads to a tree whose nodes correspond to sampling units (primary sampling units, secondary sampling units and so forth until elementary sampling units). Without loss of generality, we illustrate the tree structure involved by multistage sampling with a small example for \( L = 3 \) (Fig. 2).

In practice the tree information is given by the data. All the needed information is a label for each unit for levels \( \ell = 1, \ldots, L \), as well as the membership of each unit of level \( \ell \) within a unit of level \( \ell - 1 \). The data corresponding to our example are illustrated in Fig. 3(a).

The same data can be stored as shown in Fig. 3(b), with an access array \( M \) containing indices of arrays \( N \) and \( O \) for delimiting the levels, array \( N \) containing the unit labels, and array \( O \) containing the membership information, that is, the label of the parent unit.

For performing statistical computations we deal with in this article, at first glance it may seem necessary to access in memory the tree path for all nodes. This is for instance the case for a recursive implementation such as that of Bellhouse [8,9] and Rylett and Bellhouse [10]. In this paper, not only do we not convert the data into a recursive data structure for representing the corresponding tree (example in Fig. 2), but moreover we propose a solution where paths are used only at a preliminary step.

We manipulate a set of four arrays: (i) \( A \) containing the level of each node (between 0 for the root and \( L \) for the leafs); (ii) \( B \) containing the paths for each node; (iii) \( C \) containing the number of
Fig. 2. Example of tree for a 3-stage sampling design.

| $u_1$ | $u_{0,1}$ | $u_2$ | $u_{1, 0}$ | $u_{0, 2}$ | $u_3$ |
|-------|-----------|-------|------------|------------|-------|
| 1     | 1         | 1     | 1          | 1          | 1     |
| 2     | 1         | 2     | 1          | 2          | 1     |
| 3     | 1         | 3     | 1          | 3          | 1     |
| 4     | 2         | 4     | 2          | 4          | 2     |
| 5     | 2         | 5     | 2          | 3          | 4     |
| 6     | 2         | 6     | 2          | 3          | 5     |
| 7     | 3         | 7     | 2          | 4          | 6     |
| 8     | 3         | 8     | 3          | 4          | 7     |
| 9     | 3         | 9     | 3          | 5          | 8     |
| 10    | 3         | 9     | 3          | 5          | 9     |
| 11    | 4         | 14    | 4          | 6          | 10    |
| 12    | 4         | 14    | 4          | 6          | 10    |
| 13    | 4         | 14    | 4          | 6          | 10    |
| 14    | 4         | 14    | 4          | 6          | 10    |
| 15    | 5         | 15    | 5          | 7          | 11    |
| 16    | 5         | 15    | 5          | 7          | 11    |
| 17    | 5         | 15    | 5          | 7          | 11    |
| 18    | 6         | 18    | 6          | 8          | 12    |
| 19    | 6         | 18    | 6          | 8          | 12    |
| 20    | 6         | 18    | 6          | 8          | 12    |
| 21    | 6         | 18    | 6          | 8          | 12    |
| 22    | 7         | 22    | 7          | 9          | 13    |
| 23    | 7         | 22    | 7          | 9          | 13    |
| 24    | 7         | 22    | 7          | 9          | 13    |
| 25    | 7         | 22    | 7          | 9          | 13    |
| 26    | 8         | 26    | 8          | 10         | 14    |
| 27    | 8         | 26    | 8          | 10         | 14    |
| 28    | 8         | 26    | 8          | 10         | 14    |
| 29    | 9         | 29    | 9          | 11         | 15    |
| 30    | 9         | 29    | 9          | 11         | 15    |
| 31    | 9         | 29    | 9          | 11         | 15    |

(a) Dataset as a sparse array. (b) Dataset as dense arrays. (c) Array representation of the tree.
nodes’ children; (iv) $D$ for efficient node browsing at a given level. Sorting the arrays $A$, $B$ and $C$ is a prerequisite for subsequent treatments.

The conversion from arrays $M$ and $O$ describing the dataset is ensured by Algorithm 1 which returns (unsorted) arrays $A$ and $B$. Note that the 0-filling step at the end of the algorithm is irrelevant when computer implementation is done in a programming language where arrays are automatically initialized with 0 or when a specific syntax exists for doing such initialization in fewer instructions. Of course it is possible to convert (sorted) arrays $A$, $B$ describing the tree into a dataset using the reverse conversion Algorithm 2 which returns arrays $M$, $N$ and $O$.

Algorithm 1 Converting a dataset to the corresponding tree.

```plaintext
1: procedure ConvertDataToTree($M, O, A, B$)  
2:   for $j = M_{1,1}$ to $M_{1,2}$ do  
3:       $c_j \leftarrow j$  
4:   end for $j$  
5:   for $i = 2$ to $l$ do  
6:       $p \leftarrow i - 1$  
7:       $k \leftarrow 0$  
8:       for $j = M_{p,1}$ to $M_{p,2}$ do  
9:           $k \leftarrow k + 1$  
10:          $n_k \leftarrow 0$  
11:       end for $j$  
12:   end for $i$  
13:   for $j = M_{L,1}$ to $M_{L,2}$ do  
14:       $k \leftarrow 0_j$  
15:       $n_k \leftarrow n_k + 1$  
16:       $c_j \leftarrow n_k$  
17:   end for $j$  
18:   for $i = 0$ to $l$ do  
19:       for $j = M_{i,1}$ to $M_{i,2}$ do  
20:           $i \leftarrow i + 1$  
21:           $B_{i,j} \leftarrow c_j$  
22:           $A_i \leftarrow i$  
23:       end if $i > 1$  
24:       $p \leftarrow i - 1$  
25:       $k \leftarrow 0_j + M_{p,1} - 1$  
26:       for $m = p$ downto $1$ do  
27:           $B_{i,m} \leftarrow c_k$  
28:       end if $m > 1$  
29:       $k \leftarrow 0_k + M_{m-1,1} - 1$  
30:   end if $i$  
31:   end for $i$  
32:   end for $j$  
33:   end for $i$  
34:   $N_l \leftarrow 1$  
35:   $j \leftarrow M_{L,2} - M_{L,1} + 2$  
36:   for $\ell = L - 1$ downto $1$ do  
37:       $k \leftarrow j + M_{\ell,2} - M_{\ell,1}$  
38:       $n \leftarrow \ell + 1$  
39:       for $i = j$ to $k$ do  
40:           for $m = n$ to $\ell$ do  
41:               $B_{i,m} \leftarrow 0$  
42:           end for $m$  
43:       end for $i$  
44:       $j \leftarrow k + 1$  
45:   end for $\ell$  
46:   end for  
47:   end procedure
```

Information contained in arrays $A$ and $B$ allows generating a reordering index $T$ according to Algorithm 3. In the latter, note that the choice of the sorting algorithm is left to the discretion of the reader. Thanks to $T$, arrays $A$ and $B$ can be reordered in place using Algorithm 4. For this procedure, an upper bound ($UB$) lower than the lower bound ($LB$) for the number of columns indicates that the array to be reordered ($G$) has one dimension only. Once the reordering of $A$ and $B$ is achieved, access array $D$ is built using Algorithm 5. Note that as soon as $D$ is built, regarding the total number of nodes $N_l$
Algorithm 2 Converting a tree to the corresponding dataset.

1: procedure ConvertTreeToData

2: \( R_0 \leftarrow 1 \)

3: \( R_1 \leftarrow 2 \)

4: \( p \leftarrow 1 \)

5: for \( i = 3 \) to \( N_1 \) do

6: \( \ell \leftarrow A_i \)

7: if \( (\ell \neq p) \) then

8: \( R_{\ell} \leftarrow 1 \)

9: \( p \leftarrow \ell \)

10: end if

11: end for

12: \( R_{L+1} \leftarrow N_1 + 1 \)

13: \( k \leftarrow 0 \)

14: for \( i = 1 \) to \( L \) do

15: \( p \leftarrow \ell - 1 \)

16: \( r \leftarrow \ell + 1 \)

17: if \( (\ell > 1) \land (\ell < L) \) then

18: for \( j = r \) to \( N_1 \) do

19: \( f_j \leftarrow FALSE \)

20: end for

21: end if

22: end for

23: \( M_{L+1} \leftarrow k + 1 \)

24: \( c \leftarrow 0 \)

25: for \( i = R_k \) to \( r - 1 \) do

26: \( U \leftarrow B_i.p \)

27: \( u \leftarrow B_i \)

28: \( k \leftarrow k + 1 \)

29: \( c \leftarrow c + 1 \)

30: \( N_k \leftarrow c \)

31: \( O_k \leftarrow U \)

32: if \( (\ell > 0) \) then

33: \( B_{j,x} \leftarrow c \)

34: if \( (\ell < L) \) then

35: for \( j = r \) to \( N_1 \) do

36: if \( \neg f_j \land (B_j.p \neq U) \land (B_{j,x} = u) \) then

37: \( B_{j,x} \leftarrow c \)

38: \( f_j \leftarrow TRUE \)

39: end if

40: end for

41: end if

42: end if

43: end for

44: \( M_{L-1} \leftarrow k \)

45: end procedure

Algorithm 3 Generating a reordering index from tables of levels and paths.

1: procedure GenerateReorderingIndex

2: for \( i = 1 \) to \( N_1 \) do

3: \( R_i \leftarrow A_i \times 10^{(L+1)} \)

4: for \( \ell = 1 \) to \( L \) do

5: \( R_i \leftarrow R_i + B_{i,\ell} \times 10^{(L-\ell)} \)

6: end for

7: \( E_i \leftarrow i \)

8: end for

9: [ sorting \( R \) by increasing order

10: elements in \( E \) are permuted at the same time as those of \( R \) ]

11: for \( i = 1 \) to \( N_1 \) do

12: \( T_{E_i} \leftarrow i \)

13: end for

14: end procedure
Algorithm 4 In-place reordering of array $G$.

1: procedure ReorderTable$(N,T,LB,UB,G)$  
   $\downarrow$ indicates a one-dimension array when it is TRUE
2: $d \leftarrow (\text{UB} < \text{LB})$  
   $\uparrow$ in-place reordering of $G$
3: $k \leftarrow 1$  
   $\downarrow$ sentinel value
4: loop  
   $\downarrow$ in-place reordering of $G$
5: $i \leftarrow 0$
6: for $j = k$ to $N$ do
7:   if ($\neg I_j$) then
8:      $I_j \leftarrow \text{TRUE}$  
   $\uparrow$ marked as used
9:   if ($j \neq T_j$) then
10:      $i \leftarrow j$
11:      $k \leftarrow i + 1$
12:      [ exit from the current loop ]
13: end if
14: if ($i = 0$) then
15:   [ exit from the procedure ]
16: end if
17: if $d$ then
18:   $e \leftarrow G_i$  
   $\uparrow$ initializing a permutation cycle
19: else
20:   for $\ell = LB$ to $UB$ do
21:      $r_{\ell} \leftarrow G_{i,\ell}$  
   $\uparrow$ $r_{[LB:UB]}$ contains a row
22: end for $\ell$
23: end if
24: $m \leftarrow i$  
   $\downarrow$ sentinel value
25: repeat  
   $\uparrow$ entering in a permutation cycle
26:   $j \leftarrow T_i$
27:   if $d$ then
28:      $t \leftarrow e$
29:      $e \leftarrow G_j$
30:      $G_j \leftarrow t$
31:   else
32:      for $\ell = LB$ to $UB$ do
33:         $t \leftarrow r_{\ell}$
34:         $r_{\ell} \leftarrow G_{j,\ell}$
35:         $G_{j,\ell} \leftarrow t$
36:      end for $\ell$
37:   end if
38:   $I_j \leftarrow \text{TRUE}$  
   $\downarrow$ permuting rows
39: end if
40: $i \leftarrow j$  
   $\uparrow$ for the next permutation
41: until ($j = m$)  
   $\uparrow$ the current permutation cycle is ended
42: end loop
43: end procedure

we have of course $NI = D_{L,2}$. Hence, when $D$ is a parameter, it is not necessary to also pass NI. Finally, the number of children for each node can be computed according to Algorithm 6. For our example, the set of arrays $A$, $B$, $C$ and $D$ is illustrated in Fig. 3(c). Note that arrays $A$ and $B$ are necessary for obtaining $C$ in proper order and for building $D$ according to the simple Algorithm 5, but later on they are no longer useful for statistical computations and corresponding memory may be released.
Algorithm 5  Building an array for browsing nodes at a given level.

1:  \textbf{procedure} \text{BuildAccessArray}\(\downarrow L, N, A, D\)  
2:     \textbf{for} \(\ell = 0\) \textbf{to} \(L\) \textbf{do}  
3:         \(N_\ell \leftarrow 0\)  \hfill \(\triangleright N[0:L]\) counts the number of nodes per level 
4:     \textbf{end for} \(\ell\)  
5:     \textbf{for} \(k = 1\) \textbf{to} \(N\) \textbf{do}  
6:         \(i \leftarrow A_k\)  
7:         \(N_i \leftarrow N_i + 1\)  
8:     \textbf{end for} \(k\)  
9:     \textbf{for} \(\ell = 0\) \textbf{to} \(L\) \textbf{do}  
10:        \(k \leftarrow 0\)  \hfill \(\triangleright \text{building the access array}\)  
11:        \(D_{\ell,1} \leftarrow k + 1\)  \hfill \(\triangleright \text{first node at the current level}\)  
12:        \(k \leftarrow k + N_k\)  \hfill \(\triangleright \text{last node at the current level}\)  
13:        \(D_{\ell,2} \leftarrow k\)  
14:     \textbf{end for} \(\ell\)  
15:  \textbf{end procedure}  

Algorithm 6  Computing the number of children for all nodes.

1:  \textbf{procedure} \text{ComputeChildrenNumbers}\(\downarrow L, A, B, D, C\)  
2:     \(C_1 \leftarrow D_{1,2} = D_{1,1} + 1\)  \hfill \(\triangleright \text{root level is trivial}\)  
3:     \(p \leftarrow 1\)  \hfill \(\triangleright \text{current level}\)  
4:     \(k \leftarrow D_{2,1}\)  
5:     \textbf{for} \(i = 2\) \textbf{to} \(D_{L-1,2}\) \textbf{do}  
6:         \(i \leftarrow A_i\)  
7:         \(c \leftarrow B_{i,p}\)  
8:         \textbf{if} \((\ell \neq p)\) \textbf{then}\hfill \(\triangleright \text{changing the current level}\)  
9:             \(p \leftarrow \ell\)  
10:            \(k \leftarrow D_{\ell+1,1}\)  
11:         \textbf{end if}\hfill \(\triangleright \text{lower bound for browsing nodes at the next level}\)  
12:         \(N \leftarrow 0\)  
13:     \textbf{for} \(j = k\) \textbf{to} \(D_{\ell+1,2}\) \textbf{do}\hfill \(\triangleright \text{browsing nodes (after the root and before the leafs)}\)  
14:         \textbf{if} \((B_{j,\ell} = c)\) \textbf{then}\hfill \(\triangleright \text{node level}\)  
15:             \(N \leftarrow N + 1\)  
16:         \textbf{else}\hfill \(\triangleright \text{browsing nodes at the next level}\)  
17:             \(k \leftarrow j\)  
18:             \textbf{[exit from the current loop]}\hfill \(\triangleright \text{updating the lower bound}\)  
19:         \textbf{end if}\hfill \(\triangleright \text{browsing leafs}\)  
20:     \textbf{end for} \(j\)  
21:     \(C_i \leftarrow N\)  
22: \textbf{end for} \(i\)  
23: \textbf{for} \(i = D_{L,1}\) \textbf{to} \(D_{L,2}\) \textbf{do}\hfill \(\triangleright \text{leaf level is trivial}\)  
24:     \(C_i \leftarrow 0\)  
25: \textbf{end for} \(i\)  
26:  \textbf{end procedure}  

For testing or Monte Carlo simulation purpose it is convenient to be able to automatically generate the hierarchical nesting of units for any number of stages. This is what \textbf{Algorithm 7} does. In this paper, this algorithm is the only one which simulates the recursion associated with a tree. According to the number of stages \(L\) and an interval for random population sizes \((\text{MIN}, \text{MAX})\), it returns the total
number of nodes in the tree \((N1)\), and arrays \(A, B\) and \(C\). Next, Algorithms 3 and 4 have to be used before executing Algorithm 5. After that, as previously, memory for arrays \(A\) and \(B\) may be released.

**Algorithm 7** Generating hierarchical nesting of units on \(L\) levels.

\[
\begin{align*}
1: & \textbf{procedure } \text{GenerateTree}(L, \text{MIN, MAX, N1, A, B, C}) \\
2: & \quad i \leftarrow 0 \\
3: & \quad j \leftarrow 1 \\
4: & \quad S_{1,0} \leftarrow 1 \\
5: & \quad \textbf{for } \ell = 1 \text{ to } L \text{ do} \\
6: & \quad \quad S_{1,\ell} \leftarrow 0 \\
7: & \quad \textbf{end for } \ell \\
8: & \quad \textbf{repeat} \\
9: & \quad \quad \textbf{for } \ell = 0 \text{ to } L \text{ do} \\
10: & \quad \quad \quad r_{\ell} \leftarrow S_{j,\ell} \\
11: & \quad \quad \textbf{end for } \ell \\
12: & \quad \quad j \leftarrow j - 1 \\
13: & \quad \quad \ell \leftarrow L + 1 \\
14: & \quad \textbf{end for } m \\
15: & \quad \quad \textbf{if } (r_m = 0) \text{ then} \\
16: & \quad \quad \quad \ell \leftarrow m \\
17: & \quad \quad \quad \textbf{exit from the current loop } \} \\
18: & \quad \textbf{end if } m \\
19: & \quad i \leftarrow i + 1 \\
20: & \quad A_i \leftarrow \ell - 1 \\
21: & \quad \textbf{for } m = 0 \text{ to } L \text{ do} \\
22: & \quad \quad B_{i,m} \leftarrow r_m \\
23: & \quad \textbf{end for } m \\
24: & \quad \textbf{if } (\ell \leq L) \text{ then} \\
25: & \quad \quad \textbf{select at random a number of children } N \text{ such that } \text{MIN} \leq N \leq \text{MAX } \} \\
26: & \quad \quad \quad C_k \leftarrow N \\
27: & \quad \quad \quad \textbf{for } k = N \text{ downto } 1 \text{ do} \\
28: & \quad \quad \quad \quad r_{\ell} \leftarrow k \\
29: & \quad \quad \quad \quad j \leftarrow j + 1 \\
30: & \quad \quad \quad \quad \textbf{for } m = 0 \text{ to } L \text{ do} \\
31: & \quad \quad \quad \quad \quad S_{j,m} \leftarrow r_m \\
32: & \quad \quad \quad \quad \textbf{end for } m \\
33: & \quad \quad \textbf{end for } k \\
34: & \quad \textbf{end if } \\
35: & \quad \textbf{until } (j = 0) \\
36: & \quad N1 \leftarrow i \\
37: & \textbf{end procedure} \\
\end{align*}
\]

**Multistage sampling without replacement**

Let \(S\) be a one-dimension array containing samples sizes associated with nodes at levels \(1 \leq \ell \leq L\). Multistage sampling without replacement as formalized by recurrence formula (1) may be implemented using Algorithm 8 which generates an indicator array \(I\). As a great diversity of sampling designs may be used, details are left to the discretion of the reader. When PPSWOR is used, array \(TX\) must be computed beforehand using Algorithm 9 — which is a mere translation of recurrence formula (10) — where \(X\) is a one-dimension array corresponding to a size variable \(x\).
Algorithm 8 Selecting a sample using multistage sampling without replacement.

1: procedure GenerateSampleIndicator \( I_{L,C,D,S,TX,I} \)
2: \( I_1 \leftarrow \text{TRUE} \) \hfill \( \triangleright \) for the root
3: for \( i = 2 \) to \( D_{L,2} \) do \hfill \( \triangleright \) going down the tree from the root
4: \( I_i \leftarrow \text{FALSE} \) \hfill \( \triangleright \) \( I_{1:1} \) is the sample indicator for multistage sampling
5: end for \( i \)
6: \( h \leftarrow 1 \)
7: for \( \ell = 0 \) downto \( L - 1 \) do \hfill \( \triangleright \) browsing the tree at level \( \ell \)
8: for \( k = D_{L,1} \) to \( D_{L,2} \) do \hfill \( \triangleright \) current population size
9: \( N \leftarrow C_k \) \hfill \( \triangleright \) belongs to the sample
10: if \( I_k \) then \hfill \( \triangleright \) current sample size
11: \( n \leftarrow S_k \)
12: \( \{ \text{Apply the sampling design for the current population, generating} \)
13: the current sample indicator} \( I_{1:N} \), possibly using \( TX \}
14: for \( i = 1 \) to \( N \) do
15: \( I_{h+i} \leftarrow I_i \)
16: end for \( i \)
17: end if
18: \( h \leftarrow h + N \)
19: end for \( k \)
20: end for \( \ell \)
21: end procedure

Algorithm 9 Computing the total for all nodes in the tree.

1: procedure ComputeNodeTotals \( I_{L,C,D,X,TX} \)
2: \( i \leftarrow 0 \)
3: for \( k = D_{L,1} \) to \( D_{L,2} \) do \hfill \( \triangleright \) browsing the leafs (elementary units)
4: \( i \leftarrow i + 1 \)
5: \( TX_k \leftarrow X_i \)
6: end for \( k \)
7: for \( \ell = L - 1 \) downto \( 0 \) do \hfill \( \triangleright \) going up the tree to the root
8: \( i \leftarrow D_{L,2} \) \hfill \( \triangleright \) browsing the nodes at level \( \ell \)
9: for \( k = D_{L,1} \) to \( D_{L,2} \) do \hfill \( \triangleright \) current population size
10: \( N \leftarrow C_k \)
11: for \( m = 1 \) to \( N \) do \hfill \( \triangleright \) current population size
12: \( i \leftarrow i + 1 \)
13: \( TX_k \leftarrow TX_k + TX_i \)
14: end for \( m \)
15: end for \( k \)
16: end for \( \ell \)
17: end procedure

Statistical computations for SRSWOR at all stages

Let \( Y \) be the array corresponding to the variable of interest \( y \). In a Monte Carlo simulation, as \( y \) is known for the totality of the elementary units, Algorithm 9 applied to \( Y \) returns \( TY \). Then it is possible to apply recurrence formula (13), translated into Algorithm 10.

In practice \( Y \) contains missing values for the elementary units not in the final sample. Recurrence formulas (12) and (14) translate into Algorithms 11 and 12, respectively, where the array \( I \) corresponds to the sample indicator.
Algorithm 10 Theoretical sampling variance when all sampling designs are SRSWOR.

1: function VarianceSRSWOR(L,C,D,S,T,Y)                           \(\triangleright\) browsing the elementary units (leafs)
2:    for \(k = D_{L,1}\) to \(D_{L,2}\) do                  \(\triangleright\) v \([1:N]\) stores the variances
3:        \(v_k \leftarrow 0\)                        \(\triangleright\) going up the tree to the root
4:    end for\(k\)
5:    for \(i = L - 1\) downto 0 do                    \(\triangleright\) browsing the nodes at level \(i\)
6:        \(i \leftarrow D_{i,2}\)
7:    for \(k = D_{i,1}\) to \(D_{i,2}\) do                \(\triangleright\) current population size
8:        \(N \leftarrow C_k\)
9:    end for\(k\)
10: \(n \leftarrow S_k\)
11: \(i \leftarrow i + 1\)
12: \(Y_i \leftarrow Y_{i-1}\)
13: \(s \leftarrow 0\)
14: for \(m = 2\) to \(n\) do                        \(\triangleright\) total or value of the variable of interest
15:    \(yi \leftarrow Y_i\)
16:    \(t \leftarrow t + yi\)
17:    \(ss \leftarrow ss + 1/(m \times (m - 1)) \times (m \times yi - t)^2\) \(\triangleright\) updating formula
18:    \(s \leftarrow s + yi\)
19: end for\(m\)
20: end for\(i\)
21: \(v_k \leftarrow N^2 \times (1 - n/N) \times ss/(N - 1)/n + N/n \times s\)
22: end for\(k\)
23: end function
24: \([\text{return } v_k]\) \(\triangleright\) the theoretical sampling variance is at the root level
25: end function

Algorithm 11 Computing total estimates when all sampling designs are SRSWOR.

1: procedure ComputeNodeTotalEstimatesSRSWOR(L,C,D,S,T,Y) \(\triangleright\) browsing the elementary units (leafs)
2: \(i \leftarrow 0\)
3: for \(k = D_{L,1}\) to \(D_{L,2}\) do \(\triangleright\) belongs to the sample
4:    \(i \leftarrow i + 1\)
5:    if \(i \leq L\) then \(\triangleright\) initialization of the recurrence
6:        \(HIE_k \leftarrow Y_i\)
7:    end if
8: end for\(k\)
9: for \(i = L - 1\) downto 0 do \(\triangleright\) going up the tree to the root
10: \(h \leftarrow D_{i,2}\)
11: for \(k = D_{i,1}\) to \(D_{i,2}\) do \(\triangleright\) browsing the nodes at level \(i\)
12: \(N \leftarrow C_k\)
13: if \(k \leq L\) then \(\triangleright\) current population size
14: \(n \leftarrow 0\)
15: \(s \leftarrow 0\)
16: \(i \leftarrow 0\)
17: for \(m = 1\) to \(N\) do \(\triangleright\) belongs to the sample
18:    \(i \leftarrow i + 1\)
19:    if \(i \leq L\) then \(\triangleright\) updating current sample size
20:        \(n \leftarrow n + 1\)
21: \(s \leftarrow s + HIE_i\)
22: end if
23: end for\(m\)
24: \(HIE_k \leftarrow n/s\) \(\triangleright\) expansion estimator
25: end if
26: \(h \leftarrow h + N\)
27: end for\(k\)
28: end for\(i\)
29: end procedure

Note that, in Algorithms 10 and 12, one-pass computation of the sum of squared deviations (ss) is performed using the numerically stable updating formula proposed by Youngs and Cramer [24] (see also [25, sec. 2.4.4]).

Statistical computations for PPSWOR at all stages

Let \(X\) be the one-dimension array corresponding to size variable \(x\), positively correlated with the variable of interest \(y\). First, Algorithm 9 is applied to \(X\) for returning \(TX\). Next, unequal inclusion probabilities have to be stored. First-order inclusion probabilities are obviously stored in a one-dimension array \((P)\). For computing overall first-order inclusion probabilities \((P^*)\), recurrence formula (2) translates into Algorithm 13.
Algorithm 12  Sampling variance estimate when all sampling designs are SRSWOR.

1: function VarianceEstimateSRSWOR(L, C, D, TY, I) ▷ browsing the elementary units (leafs)
2: for k = D_L,1 to D_L,2 do ▷ v[1:N1] stores the variances
3: \( v_k \leftarrow 0 \)
4: end for k
5: for \( \ell = L - 1 \) downto 0 do ▷ going up the tree to the root
6: \( h \leftarrow D_{\ell,2} \)
7: for \( k = D_{\ell,1} \) to \( D_{\ell,2} \) do ▷ browsing the nodes at level \( \ell \)
8: \( N \leftarrow C_k \)
9: if \( I_k \) then ▷ current population size
10: \( n \leftarrow 0 \) ▷ belongs to the sample
11: ss \( \leftarrow 0 \)
12: \( i \leftarrow h \)
13: end if
14: for \( m = 1 \) to \( N \) do ▷ belongs to the sample
15: \( i \leftarrow i + 1 \)
16: if \( l_i \) then
17: \( n \leftarrow n + 1 \) ▷ initialization
18: \( t \leftarrow TY_i \) ▷ total
19: ss \( \leftarrow 0 \)
20: \( s \leftarrow v_i \)
21: else ▷ total
22: \( yi \leftarrow TY_i \)
23: \( t \leftarrow t + yi \)
24: ss \( \leftarrow ss + 1/(n \times (n - 1)) \times (n \times yi - t)^2 \) ▷ updating formula
25: \( s \leftarrow s + v_i \)
26: end if
27: end for m
28: \( v_k = N^2 \times (1 - n/N) \times ss/(n - 1)/n + N/n \times s \) ▷ the sampling variance estimate is at the root level
29: end if
30: \( h \leftarrow h + N \)
31: end for k
32: end for \( \ell \)
33: return \( v_1 \)
34: end function

Algorithm 13  Computing overall first-order inclusion probabilities.

1: procedure ComputeOverallProbabilities(L, C, D, P, \( \pi^* \)) ▷ \( \pi^* = 1 \) for the root node
2: \( P^*_1 \leftarrow 1 \)
3: \( i \leftarrow 1 \)
4: for \( \ell = 0 \) to \( L - 1 \) do ▷ going down the tree from the root
5: for \( k = D_{\ell,1} \) to \( D_{\ell,2} \) do ▷ browsing the nodes at level \( \ell \)
6: \( N \leftarrow C_k \)
7: for \( j = 1 \) to \( N \) do ▷ number of children for node \( k \)
8: \( i \leftarrow i + 1 \)
9: \( P^*_k \leftarrow P_i \times P^*_k \)
10: end for j
11: end for k
12: end for \( \ell \)
13: end procedure
When joint inclusion probabilities can be managed, they have to be stored in an efficient way, that is in a one-dimension array. For the complete set of units involved in a multistage sampling situation, the matrix of joint inclusion probabilities is block diagonal. Owing to the symmetry of joint inclusion probabilities and to the block diagonal structure, it turns out that a very efficient storage requires only two one-dimension arrays: (i) a small one ($F$) for indicating submatrix changes; (ii) a larger one ($IP$) to store all joint inclusion probabilities (without duplication). These arrays are generated using Algorithm 14. When joint inclusion probabilities cannot be managed, $F$ and $IP$ arrays are irrelevant, and Algorithm 14 has to be simplified accordingly (this straightforward modification if left to the reader). Since a great diversity of sampling designs may be used, parameters required for computing $\Pi$ matrices are omitted and details are left to the discretion of the reader. Finally, retrieving joint inclusion probabilities needs Algorithm 15 which, in turn, relies on Algorithm 16.

**Algorithm 14** Storage of first- and second-order inclusion probabilities.

1: `procedure StoreInclusionProbabilities(L,C,D,JX,P,F,IP)`
2: \[ \text{\textit{i} \leftarrow 0} \]
3: \[ \text{\textit{p} \leftarrow 1} \]
4: \[ \text{\textbf{for} } \text{\textit{e} = 0} \text{\textbf{to} } L - 1 \text{ \textbf{do}} \]
5: \[ \text{\textbf{for} } \text{\textit{k} = } D_{\textit{e},1} \text{ \textbf{to} } D_{\textit{e},2} \text{ \textbf{do}} \]
6: \[ \text{\textit{i} \leftarrow \textit{i} + 1} \]
7: \[ \text{\textit{F}_1 \leftarrow \textit{p} \]} \]
8: \[ \text{\textit{N} \leftarrow C_k} \]
9: \[ \text{\textit{m} \leftarrow N \times (N - 1)/2} \]
10: \[ \text{\textit{p} \leftarrow \textit{p} + \textit{m} \]} \]
11: \[ \text{\textbf{end for} } \textit{k} \]
12: \[ \text{\textbf{end for} } \textit{e} \]
13: \[ \text{\textit{N}_2 \leftarrow \textit{p} - 1} \]
14: \[ \text{\textit{m} \leftarrow 1} \]
15: \[ \text{\textit{P}_1 \leftarrow 1} \]
16: \[ \text{\textit{n} \leftarrow 0} \]
17: \[ \text{\textbf{for} } \text{\textit{e} = 0} \text{ \textbf{to} } L - 1 \text{ \textbf{do}} \]
18: \[ \text{\textbf{for} } \text{\textit{k} = } D_{\textit{e},1} \text{ \textbf{to} } D_{\textit{e},2} \text{ \textbf{do}} \]
19: \[ \text{\textbf{[ Compute the matrix } \Pi \text{ for current population and design, }} \]
20: \[ \text{\textbf{possibly using } JX} \]
21: \[ \text{\textit{N} \leftarrow C_k} \]
22: \[ \text{\textbf{for} } \text{\textit{i} = 1} \text{ \textbf{to} } \textit{N} \text{ \textbf{do}} \]
23: \[ \text{\textit{m} \leftarrow \textit{m} + 1} \]
24: \[ \text{\textit{P}_m \leftarrow } \Pi_{i,i} \]
25: \[ \text{\textbf{if} } (\textit{N} > \textit{i}) \text{ \textbf{then}} \]
26: \[ \text{\textbf{for} } \text{\textit{j} = i + 1} \text{ \textbf{to} } \textit{N} \text{ \textbf{do}} \]
27: \[ \text{\textit{n} \leftarrow n + 1} \]
28: \[ \text{\textit{IP}_n \leftarrow } \Pi_{i,j} \]
29: \[ \text{\textbf{end for} } \textit{j} \]
30: \[ \text{\textbf{end if} } \]
31: \[ \text{\textbf{end for} } \textit{i} \]
32: \[ \text{\textbf{end for} } \textit{k} \]
33: \[ \text{\textbf{end for} } \textit{e} \]
34: \[ \text{\textbf{end procedure} } \]

When Monte Carlo simulations are performed, computing the theoretical sampling variance in the general case (that is, with unequal inclusion probabilities) requires recurrence formula (4) translated into Algorithm 17. Exact computation needs formula (5) (Sen-Yates-Grundy variance) which translates into Algorithm 18 and returns array $V$ required by Algorithm 17.
Algorithm 15  Returning the joint inclusion probability for \( i \neq j \) in the subpopulation of size \( N \) corresponding to the node \( k \).

1: function \( \text{PI}(F, IP, N, i, j, k) \)
2: \[ e \leftarrow F_k + \text{SymmetricMatrixIndex}(N, i, j) - 1 \]
3: \[ \text{return } IP_e \]
4: end function

Algorithm 16  Returning the index for an element in a symmetric matrix which upper triangular part is stored in an one-dimension array.

1: function \( \text{SymmetricMatrixIndex}(n, k, \ell) \)  
2: \[ \text{if } (\ell > k) \text{ then} \]
3: \[ i \leftarrow k \]
4: \[ j \leftarrow \ell \]
5: \[ \text{else} \]
6: \[ i \leftarrow \ell \]
7: \[ j \leftarrow k \]
8: \[ \text{end if} \]
9: \[ e \leftarrow 0 \]
10: \[ \text{if } (i > 1) \text{ then} \]
11: \[ \text{for } m = 1 \text{ to } i - 1 \text{ do} \]
12: \[ e \leftarrow e + n - m \]
13: \[ \text{end for } m \]
14: \[ \text{end if} \]
15: \[ e \leftarrow e + j - i \]
16: \[ \text{return } e \]
17: end function

Unbiased estimation of the population total is achieved using recurrence formula (3), translated into Algorithm 19. Estimating the sampling variance requires recurrence formula (6) translated into Algorithm 20. When joint inclusion probabilities can be managed, node variance estimates can be computed using formula (7), translated into Algorithm 21 returning \( V \) which is a parameter of Algorithm 20. Otherwise, one possibility is to use approximation (8) translated into Algorithm 22, which only requires first-order inclusion probabilities.

Method validation

This section is aimed at providing both numerical illustration and control for the validity of statistical computation algorithms proposed in this article.

Generating artificial multistage sampling configurations for any number of stages (Algorithm 7) is a very convenient way for applying and validating all algorithms proposed in this article on very large examples. Nevertheless, there is no generality loss by referring to the small example used throughout this paper, with \( L = 3 \) stages and \( \text{MIN} = 3 \) or \( \text{MAX} = 4 \) children for each node at levels \( 1 \leq \ell \leq L \).

Self-weighted multistage sampling design is a good candidate for testing purpose since it involves unequal probability sampling. Besides, the self-weighting property is very easy to check thanks to selection equation (2); an error in this regard would immediately indicate an implementation error. As PPSWOR design we refer here to the Hanurav-Vijayan procedure which is appropriate, provided it is correctly implemented [22]. For the sake of completeness, we also provide the explicit expression of the theoretical sampling variance (20), which translates into Algorithm 23. This example shows how cumbersome the expression of the variance becomes when the number of stages increases. Terms \( s_{1,1}, s_{1,2}, s_{2,1} \) and \( s_{2,2} \) are indicated to easily identify their counterparts in Algorithm 23.
\[
V(\bar{Y}) = \sum_{i<j} \left( \pi_{1,i} \pi_{1,j} - \pi_{1,ij} \right) \left( \frac{Y_{1,i}}{\pi_{1,i}} - \frac{Y_{1,j}}{\pi_{1,j}} \right) + \sum_{i \in h} \frac{1}{n_i} \sum_{k \in c_{d_k}} \left( \pi_{2,k} \pi_{2,\ell} - \pi_{2,k,\ell} \right) \left( \frac{Y_{2,k}}{\pi_{2,k}} - \frac{Y_{2,\ell}}{\pi_{2,\ell}} \right) \left( \sum_{k \in c_{d_k}} \frac{1}{n_{2,k}} \left[ N_{3,k}^2 \left( 1 - \frac{n_{3,k}}{N_{3,k}} \right) \right] \right) \]
(20)

**Algorithm 17** Theoretical sampling variance in the general case.

1: function Variance(L, C, D, P, V) \(\triangleright\) browsing the elementary units (leaves)  
2: for \(k = D_{l,1}\) to \(D_{l,2}\) do \(\triangleright\) \(v[1:N]\) stores the variances  
3: \(v_k \leftarrow 0\) \(\triangleright\) going up the tree to the root  
4: end for \(k\)  
5: for \(\ell = L - 1\) downto 0 do \(\triangleright\) browsing the nodes at level \(\ell\)  
6: \(i \leftarrow D_{l,2}\) \(\triangleright\) current population size  
7: for \(k = D_{l,1}\) to \(D_{l,2}\) do  
8: \(N \leftarrow C_k\)  
9: \(s \leftarrow 0\)  
10: for \(m = 1\) to \(N\) do  
11: \(i \leftarrow i + 1\)  
12: \(s \leftarrow s + v_i / P_i\)  
13: end for \(m\)  
14: \(v_k \leftarrow V_k + s\)  
15: end for \(k\)  
16: end for \(\ell\)  
17: [return \(v_1\)] \(\triangleright\) the theoretical sampling variance is at the root level  
18: end function

**Algorithm 18** Computing node variances (Sen-Yates-Grundy).

1: procedure ComputeNodeVarianceSYG(L, C, D, P, \(\pi\))  
2: \(h = 1\) \(\triangleright\) going down the tree from the root  
3: for \(\ell = 0\) to \(L - 1\) do \(\triangleright\) browsing the nodes at level \(\ell\)  
4: for \(k = D_{l,1}\) to \(D_{l,2}\) do \(\triangleright\) current population size  
5: \(N \leftarrow C_k\)  
6: \(s = 0\)  
7: for \(m = 1\) to \(N - 1\) do  
8: \(j \leftarrow j + 1\)  
9: \(P_i \leftarrow P_j\) \(\triangleright\) \(\pi_j\) \(\triangleright\) total \(\triangleright\) \(\pi\)-expanded value  
10: \(y_i \leftarrow Y_i\)  
11: \(z_i \leftarrow y_i / P_i\)  
12: \(j \leftarrow j - 1\) \(\triangleright\) \(\pi_j\) \(\triangleright\) total \(\triangleright\) total \(\triangleright\) \(\pi\)-expanded value \(\triangleright\) unpacked \(\pi_{i,j}\)  
13: for \(n = m + 1\) to \(N\) do  
14: \(j \leftarrow j + 1\)  
15: \(P_j \leftarrow P_j\)  
16: \(y_j \leftarrow Y_j\)  
17: \(z_j \leftarrow y_j / P_j\)  
18: \(P_{ij} \leftarrow P_i / (P_i \times P_j)\) \(\triangleright\) \(V[1:N]\) is the array of variances  
19: \(s \leftarrow s + (z_j - z_i)^2 \times (P_i \times P_j - P_{ij})\)  
20: end for \(n\)  
21: end for \(m\)  
22: end for \(\ell\)  
23: \(V_k \leftarrow s\) \(\triangleright\) \(V[1:N]\)  
24: \(h \leftarrow h + N\) \(\triangleright\) \(V[1:N]\)  
25: end for \(k\)  
26: end for \(\ell\)  
27: end procedure
Theoretical first-order ($P$) and overall ($P^*$) inclusion probabilities as well as estimates of the latter ($\hat{P}^*$) obtained after $10^7$ replications of sampling [22, sec. 4.1] are given in Fig. 4(a). An example of sample is given in column I (sample indicator) and the result of procedure 19 appears in column HIE (the total estimate is $\hat{Y} = 15.655$). The sampling variance computed by function 17 is $V(\hat{Y}) = 7.875$. As expected this is also the value obtained using Algorithm 23. We can estimate the sampling variance using either Sen-Yates-Grundy variance estimator (7) (case 1) or approximation (8) (case 2). In the former case, computations require second-order inclusion probabilities given in Fig. 4(b). Sampling variance estimates for the sample given in 4(a) are $\hat{V}(\hat{Y}) = 18.945$ for case 1 and $\hat{V}(\hat{Y}) = 13.931$ for case 2. As expected, the Monte Carlo estimate for the sampling variance is close to its theoretical value since we obtain 7.870. Finally, the Monte Carlo estimate for the expectation of $\hat{V}(\hat{Y})$ is 7.873 for case 1 — again very close to the theoretical value — and 7.652 for case 2, which indicates a slight underestimation due to the approximation, and illustrates that estimator (8) tends to be down-biased.
Fig. 4. Numerical illustration for the 3-stage self-weighted sampling design used as an example. (a) Summary table with some data and results: $i$ - node index; $A$ - node level; $C$ - children number; $n$ - size sample; $TX$ - node size; $TY$ - node total; $P$ - inclusion probability, $P^*$ - overall inclusion probability; $\hat{P}^*$ - Monte Carlo estimate of $P^*$ ($10^7$ replications); $I$ - HTE - expansion estimator. (b) Array representation of second-order inclusion probabilities: $F$ - index indicating submatrix changes; $PP$ - joint inclusion probabilities.
Algorithm 21 Computing Sen-Yates-Grundy variance estimates.

1: procedure ComputeNodeVarianceEstimatesSYG(N, ν, k, ℓ, Pᵢ, Pⱼ, D_k, Dₜ, C_k, Cₜ)
2:   h ← 1
3:   for i = 0 to ℓ do
4:     if k = Dₜ+1 to Dₜ+2 do
5:       N ← Cₜ
6:         if h then
7:           s ← 0
8:           j ← h
9:         for m = 1 to N do
10:           i ← i + 1
11:           if h then
12:             Pᵢ ← P₁
13:             yᵢ ← HTEᵢ
14:             zᵢ ← yᵢ/Pᵢ
15:             j ← j + 1
16:             for n = m + 1 to N do
17:               j ← j + 1
18:               if h then
19:                 Pⱼ ← P₂
20:                 yⱼ ← HTEⱼ
21:                 zⱼ ← yⱼ/Pⱼ
22:                 (Pⱼ − Pᵢ) − HTEᵢ)
23:                 s ← s + (zi − zij)² × (Pᵢ × Pⱼ − PᵢPⱼ)/ν
24:               end if
25:           end for n
26:         end if
27:       end for m
28:     end if
29:   end for i
30: end procedure

Additional information

In practice, two-stage and three-stage sampling designs are rather common, and four-stage sampling is not exceptional. Conversely, multistage sampling beyond five stages seems not to be used.

When sampling geographical space, multistage sampling is very useful from a logistical point of view for reducing the cost of travelling between the elementary spatial sampling units. It follows that, in environmental sciences for instance, multistage sampling is part of the toolbox for designing large-scale natural resource surveys, often in combination with stratification. The multistage sampling of nested spatial sampling units also meets the concern of being able to study a phenomenon at several spatial scales.

The data collected on the elementary units are not always devoted only to estimating finite population parameters, and may rather be seen as all-purpose data. As such, their analysis and modeling should be easy, which implies they all have the same weight, that is, the same overall inclusion probability. Otherwise, specific versions of the usual statistics methods should be used. It is always risky to communicate data that one cannot process correctly using classical statistics methods. Hence, for avoiding this risk, it is advisable that the multistage sampling design be a self-weighting one. It is worth noting that self-weighted multistage sampling also allows balancing workload among the high-level units, which, again, may be interesting from a logistical point of view.
Algorithm 22 Computing approximate variance estimates.

1: procedure ComputeNodeVarianceEstimates(T, L, C, D, S, HTE, P, I, V)
1
2: \( h \leftarrow 1 \)
3: for \( i = 0 \) to \( L - 1 \) do
4: \( \text{for } k = D_L, 1 \text{ to } D_L, 2 \text{ do} \)
5: \( N \leftarrow C_k \)
6: \( \text{if } I_k \text{ then} \)
7: \( \text{for } m = 1 \text{ to } N \text{ do} \)
8: \( i \leftarrow i + 1 \)
9: \( j \leftarrow 0 \)
10: \( \text{if } I_j \text{ then} \)
11: \( P_j \leftarrow P_i \)
12: \( c_j = \frac{(1 - P_j) \times a}{u} \)
13: \( u \leftarrow u + c_j \)
14: \( y_j \leftarrow HTE_i \) \( \triangleright \) total estimate or value of the variable of interest
15: \( y_j^* \leftarrow P_j \times b \)
16: \( s \leftarrow s + c_j \times (y_j - y_j^*)^2 \)
17: \( \text{end if} \)
18: \( \text{end for } m \)
19: \( V_k \leftarrow s \)
20: \( \text{end if} \)
21: \( h \leftarrow h + N \)
22: \( \text{end for } k \)
23: \( \text{end for } i \)
24: \( \text{end procedure} \)
Algorithm 23  Explicit variance computation for self-weighted three-stage sampling.

1: function Variance3StagesEPSEM(C, D, S, Y, P, F, P')
2:     N1 = C₁
3:     i₁ = 1
4:     s₁₁ = 0
5:     for i = 1 to N1 - 1 do
6:         i₁ = i₁ + 1
7:         Pᵢ = Pᵢ₁
8:         yᵢ = Yᵢ₁
9:         zᵢ = yᵢ/Pᵢ
10:        j₁ = i₁
11:     for j = i₁ + 1 to N1 do
12:         j₁ = j₁ + 1
13:         Pⱼ = Pⱼ₁
14:         yⱼ = Yⱼ₁
15:         zⱼ = yⱼ/Pⱼ
16:        Pⱼ₁ = Pⱼ₁(F, F', N₁, i, j, k)
17:     s₁₁ = s₁₁ + (zⱼ - z₁)² × (Pᵢ × Pⱼ₁ - Pᵢ₁)
18:     end for j
19:     end for i
20:     h₁₂ = D₁₂
21:     h₁₁ = D₁₁
22:     for k = D₁₁ to D₁₂ do
23:     N₂ = C₂
24:     P_k = P_k₁
25:     s₂₁ = 0
26:     s₂₂ = 0
27:     i₂ = h₁₂
28:     for i = 1 to N₂ do
29:         i₂ = i₂ + 1
30:         Pᵢ = Pᵢ₂
31:         yᵢ = Yᵢ₂
32:         zᵢ = yᵢ/Pᵢ
33:         if (i < N₂) then
34:             i₂ = i₂
35:         end if
36:     j₂ = i₂ + 1
37:         Pⱼ = Pⱼ₂
38:         yⱼ = Yⱼ₂
39:         zⱼ = yⱼ/Pⱼ
40:        Pⱼ₁ = Pⱼ₁(F, F', N₂, i, j, k)
41:     s₂₁ = s₂₁ + (zⱼ - z₁)² × (Pᵢ × Pⱼ₁ - Pᵢ₁)
42:     end for j
43:     end if
44:     N₃ = C₃
45:     n = S₁₂
46:     i₃ = i₁ + 1
47:     t = Yᵢ₁
48:     s₃ = 0
49:     for j = 2 to N₃ do
50:         i₃ = i₃ + 1
51:         yᵢ = Yᵢ₃
52:         t = t + yᵢ
53:         s₃ = s₃ + 1/(j × (j - 1)) × (j × yᵢ - pt)²
54:     end for j
55:     s₃₂₂ = s₃₁₂ + (s₂₁(N₁ - 1)/n × N₃² × (1 - n/N₁))/Pᵢ
56:     h₃ = h₃₁ + N₃
57:     for i = 1
58:         s₁₁ = s₁₁ + (s₂₁ + s₂₂)/P_k
59:     h₂ = h₂ + N₂
60:     end for k
61:     return s₁₁ + s₁₂
62: end function
Declaration of Competing Interest

The author declare that he has no known competing for financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Many thanks to my colleague Matthieu Guillemain for checking the English of this article.

References

[1] J. Durbin, Some results in sampling theory when the units are selected with unequal probabilities, J. Roy. Stat. Soc. B 15 (1953) 262–269, doi:10.1111/j.2517-6161.1953.tb00141.x.
[2] D. Raj, Some remarks on a simple procedure of sampling without replacement, J. Am. Stat. Assoc. 61 (1966) 391–396, doi:10.1080/01621459.1966.10480874.
[3] J. Rao, Unbiased variance estimation for multistage designs, Sankhyā C 37 (1975) 133–139.
[4] C. Särndal, B. Swensson, J. Wretman, Model assisted survey sampling, Springer, New York, 1992.
[5] Y. Tillé, Sampling and estimation from finite populations, John Wiley & Sons, Hoboken, NJ, 2020.
[6] J. Rohl, Recursion via Pascal, Cambridge University Press, Cambridge, UK, 1984.
[7] M. Rubio-Sánchez, Introduction to recursive programming, CRC Press, Boca Raton, FL, 2018.
[8] D. Bellhouse, Computation of variance-covariance estimates for general multistage sampling designs, in: M.M. Barritt, D. Wishart (Eds.), COMPSTAT 1980: Proceedings in computational statistics. 4th symposium held at edinburgh 1980, Physica-Verlag, Vienna, Austria, 1980, pp. 57–63.
[9] D. Bellhouse, Computing methods for variance estimation in complex surveys, J. Off. Stat. 1 (1985) 323–329.
[10] D. Rylett, D. Bellhouse, TREES: a computer program for complex surveys, in: Proceedings of the survey research methods section, ASA, session XIX, sampling error: Methodology, software and applications, American Statistical Association, Alexandria, VA, 1988, pp. 694–697.
[11] N. Caron, Le logiciel POULPE : aspects méthodologiques, in: INSEE (Ed.), Actes des journées de méthodologie statistique - 17 et et 18 mars 1998. INSEE méthodes n° 84-85-86, INSEE, Paris, France, 1999, pp. 173–199.
[12] J. Petit, Le logiciel POULPE : modélisation informatique, in: INSEE (Ed.), Actes des journées de méthodologie statistique - 17 et et 18 mars 1998. INSEE méthodes n° 84-85-86, INSEE, Paris, France, 1999, pp. 201–220.
[13] C. Goga, Brief review of survey sampling techniques with R, Rom. Stat. Rev. 66 (2018) 83–94.
[14] Y. Tillé, A. Matei, Teaching survey sampling with the ‘sampling’ R package, in: C. Reading (Ed.), ICOTS-8 conference proceedings, data and context in statistics education: towards an evidence-based society, International Association for Statistical Education, Kuala Lumpur, Malaysia, 2010.
[15] T. Lumley, Analysis of complex survey samples, J. Stat. Soft. 9 (2004) 1–19, doi:10.18637/jss.v009.i08.
[16] T. Lumley, Complex surveys. A guide to analysis using R, John Wiley & Sons, Hoboken, NJ, 2010.
[17] D. Zardetto, ReGeneses: and advanced R system for calibration, estimation and sampling error assessment in complex sample surveys, J. Off. Stat. 31 (2015) 177–203, doi:10.1515/jos-2015-0013.
[18] A. Hedayat, B. Sinha, Design and inference in finite population sampling, John Wiley & Sons, New York, 1991.
[19] C. Daffin, P. Duncombe, M. Hills, P. North, G. Wetherill, A microcomputer survey analysis package designed for the developing countries, The Statistician 35 (1986) 505–523, doi:10.2307/2987966.
[20] L. Kish, Survey sampling, John Wiley & Sons, New York, USA, 1965.
[21] G. Kalton, Introduction to survey sampling, Sage Publications, Newbury Park, CA, 1983.
[22] P. Aubry, On the correct implementation of the Hanurav-Vijayan selection procedure for unequal probability sampling without replacement, Comm. Stat. - Simul. Comput. (2021), doi:10.1080/03610918.2021.1891431.
[23] A. Matei, Y. Tillé, Evaluation of variance approximations and estimators in maximum entropy sampling with unequal probability and fixed sample size, J. Off. Stat. 21 (2005) 543–570.
[24] E. Youngs, E. Cramer, Some results relevant to choice of sum and sum-of-product algorithm, Technometrics 13 (1971) 657–665, doi:10.1080/00401706.1971.10488826.
[25] C. Weihls, O. Mersmann, U. Ligges, Foundations of statistical algorithms. With references to R packages, CRC Press, Boca Raton, Florida, USA, 2014.