An Object-Oriented Framework for Statistical Simulation: The R Package simFrame

Andreas Alfons
Erasmus Universiteit Rotterdam

Matthias Templ
Vienna University of Technology, Statistics Austria

Peter Filzmoser
Vienna University of Technology

Abstract

This package vignette is a modified version of Alfons *et al.* (2010d), published in the *Journal of Statistical Software*.

Simulation studies are widely used by statisticians to gain insight into the quality of developed methods. Usually some guidelines regarding, e.g., simulation designs, contamination, missing data models or evaluation criteria are necessary in order to draw meaningful conclusions. The R package **simFrame** is an object-oriented framework for statistical simulation, which allows researchers to make use of a wide range of simulation designs with a minimal effort of programming. Its object-oriented implementation provides clear interfaces for extensions by the user. Since statistical simulation is an embarrassingly parallel process, the framework supports parallel computing to increase computational performance. Furthermore, an appropriate plot method is selected automatically depending on the structure of the simulation results. In this paper, the implementation of **simFrame** is discussed in great detail and the functionality of the framework is demonstrated in examples for different simulation designs.

Keywords: R, statistical simulation, outliers, missing values, parallel computing.

1. Introduction

Due to the complexity of modern statistical methods, obtaining analytical results about their properties is often virtually impossible. Therefore, simulation studies are widely used by statisticians as data-based, computer-intensive alternatives for gaining insight into the quality of developed methods. However, research projects commonly involve many scientists, often from different institutions, each focusing on different aspects of the project. If these researchers use different simulation designs, the results may be incomparable, which in turn makes it impossible to draw meaningful conclusions. Hence simulation studies in such research projects require a precise outline.

The R package **simFrame** (Alfons 2013) is an object-oriented framework for statistical simulation addressing this problem. Its implementation follows an object-oriented approach based on S4 classes and methods (Chambers 1998, 2008). A key feature is that statisticians can make use of a wide range of simulation designs with a minimal effort of programming. The object-oriented implementation gives maximum control over input and output, while at the same time providing clear interfaces for extensions by user-defined classes and methods.

Comprehensive literature exists on statistical simulation, but is mainly focused on technical aspects (e.g., Morgan 1984; Ripley 1987; Johnson 1987). Unfortunately, hardly any publications are available regarding the conceptual elements and general design of modern simulation experiments. To name some examples, Münnich *et al.* (2003) and Alfons *et al.* (2009) describe how close-to-reality simulations may be performed in survey statistics, while Burton *et al.* (2006) address applications in medical statistics. Furthermore, while simulation studies are widely used in scientific articles, they are often described only briefly and without sufficient details on all the processes involved. Having a framework with different simulation designs ready at hand may help statisticians to plan simulation studies for their needs.

Statistical simulation is frequently divided into two categories: *design-based* and *model-based* simulation. Design-based simulation is popular in survey statistics, as samples are drawn repeatedly from a finite population. The close-to-reality approach thereby uses the true sampling designs for real-life surveys such as EU-SILC (European Union Statistics on Income and Living Conditions). In every iteration, certain estimators such as indicators are computed or other statistical procedures such as imputation are applied. The obtained values can then be compared to the true population values where appropriate. Nevertheless, since real population data is only in few cases available to researchers, synthetic populations may be generated from existing samples (see, e.g., Münnich *et al.* 2003; Münnich and Schürle 2003; Raghunathan *et al.* 2003; Alfons *et al.* 2010b). Such synthetic populations must reflect the structure of the underlying sample regarding dependencies among the variables and heterogeneity. For household surveys, population data can be generated using the R package **simPopulation** (Alfons and Kraft 2010). In model-based simulation, on the other hand, data sets are generated repeatedly from a distributional model or a mixture of distributions. In every iteration, certain methods are applied and quantities of interest are computed for comparison. Where appropriate, reference values can be obtained from the underlying theoretical distribution. *Mixed* simulation designs constitute a combination of the two approaches, in which samples are drawn repeatedly from each generated data set.

The package **simFrame** is intended to be as general as possible, but has initially been developed for close-to-reality simulation studies in survey statistics. Moreover, it is focused on simulations involving typical data problems such as outliers and missing values. Therefore, certain proportions of the data may be contaminated or set as missing in order to investigate the quality and behavior of, e.g., robust estimators or imputation methods. In addition, an appropriate plot method for the simulation results is selected automatically depending on their structure. Note that statistical simulation is a very loose concept, though, and that the application of **simFrame** may be subject to limitations in certain scenarios.

Section 2 gives a brief introduction to the basic concepts of object-oriented programming and the S4 system. In Section 3, the design of the framework is motivated and Section 4 describes the implementation in great detail. Section 5 then provides details about parallel computing with **simFrame**. The use of the package for different simulation designs is demonstrated in Section 6. Additional examples for design-based simulation are given in a supplementary paper. How to extend the framework is outlined in Section 7. Finally, Section 8 contains concluding remarks and gives an outlook on future developments.

2. Object-oriented programming and S4

The object-oriented paradigm states that problems are formulated using interacting objects rather than a set of functions. The properties of these objects are defined by *classes* and their behavior and interactions are modeled with *generic functions* and *methods*. One of the most important concepts of object-oriented programming is *class inheritance*, i.e., *subclasses* inherit properties and behavior from their *superclasses*. Thus code can be shared for related classes, which is the main advantage of inheritance. In addition, subclasses may have additional properties and behavior, so in this sense they *extend* their superclasses. In S4 (Chambers 1998, 2008), properties of objects are stored in *slots* and can be accessed or modified with the `@` operator or the `slot()` function. However, *accessor* and *mutator* methods are supposed to be used to access or modify properties of objects in **simFrame** (see Section 3.3). *Virtual classes* are special classes from which no objects can be created. They exist for the sole reason of sharing code. Furthermore, *class unions* are special virtual classes with no slots.

Generic functions define the formal arguments that are evaluated in a function call in order to select the actual method to be used. These methods are defined by their *signatures*, which assign classes to the formal arguments. In short, generic functions define *what* should be done and methods define *how* this should be done for different (combinations of) classes. As an example, the generic function `setNA()` is used in **simFrame** to insert missing values into a data set. These are the available methods:

```
R> showMethods("setNA")

Function: setNA (package simFrame)
x="data.frame", control="NAControl"
x="data.frame", control="character"
x="data.frame", control="missing"
```

Even though a simple object-oriented mechanism was introduced in S3 (Chambers and Hastie 1992), it is not sufficient for the purpose of implementing a flexible framework for statistical simulation. Only S4 offers consequent implementations of advanced object-oriented techniques such as inheritance, object validation and method signatures. In S3, inheritance is realized by simply using a vector for the `class` attribute, hence there is no way to guarantee that the subclass contains all properties of the superclass. It should be noted that the tradeoff of these advanced programming techniques is a slightly increased computational overhead. Nevertheless, with modern computing power, this is not a substantial issue.

3. Design of the framework

Statistical simulation in R (R Development Core Team 2010) is often done using bespoke use-once-and-throw-away scripts, which is perfectly fine when only a handful of simulation studies need to be done for a specific purpose such as a paper. But when a research project is based on extensive simulation studies with many different simulation designs, this approach has its limitations since substantial changes may need to be applied to the R scripts for each design. In addition, if many partners are involved in the project and each of them writes their own scripts, they need to be very well coordinated so that the implemented simulation designs are similar, otherwise the obtained results may not be comparable.

The fundamental design principle of **simFrame** is that the behavior of functions is determined by *control objects*. A collection of such control objects, including a function to be applied in each iteration, is simply plugged into a generic function called `runSimulation()`, which then performs the simulation experiment. This allows to easily switch from one simulation design to another by just plugging in different control objects. Note that the user does not have to program any loops for iterations or collect the results in a suitable data structure, the framework takes care of this. Furthermore, by using the package as a common framework for simulation in research projects, guidelines for simulation studies may be defined by selecting specific control classes and parameter settings. If the researchers decide on a set of control objects to be used in the simulation studies, this ensures comparability of the obtained results and avoids problems with drawing conclusions from the project. Defining control objects thereby requires only a few lines of code, and storing them as `RData` files in order to distribute them among partners is much easier than ensuring that a large number of R scripts with big chunks of bespoke code are comparable.

As a motivational example, consider a research project in which researchers A and B investigate a specific survey such as EU-SILC (European Union Statistics on Income and Living Conditions). Researcher A focuses on robust estimation of certain indicators, while researcher B tries to improve the data quality with more suitable imputation and outlier detection methods. The aim of the project is to evaluate the developed methods with extensive simulation studies. In order to be as realistic as possible, design-based simulation studies are performed, where samples are drawn repeatedly from (synthetic) population data. Let the survey of interest in real life be conducted in many countries with different sampling designs. Then A and B could each define some control objects for the most common sampling designs and exchange them so that they can plug each of them into the function `runSimulation()` along with the population data.

Since imputation methods and outlier detection methods typically make some theoretical assumptions about the data, B could also carry out model-based simulation studies, in which the data are repeatedly generated from a certain theoretical distribution. All B needs to change is to define a control object to generate the data and supply it to `runSimulation()` instead of the population data and the control object for sampling.

Both researchers in this example investigate robust methods. It may be of interest to explore the behavior of these methods under different contamination models (the term *contamination* is used in a technical sense in this paper, see Section 4.3 for a definition). This can again be done by defining and exchanging a set of control objects. In addition, B can define various control objects for inserting missing values into the data in order to study the performance of imputation methods. Switching from one contamination model or missing data mechanism to another is simply done by replacing the respective control object in the call to `runSimulation()`. B could also supply a control object for inserting contamination and one for inserting missing values to investigate robust imputation methods or outlier detection methods for incomplete data.

One example for such research projects is the project AMELI (Advanced Methodology of European Laeken Indicators, `http://ameli.surveystatistics.net`), in the course of which the package **simFrame** has been developed.

3.1. UML class diagram

The Unified Modeling Language (UML) (Fowler 2003) is a standardized modeling language used in software engineering. It provides a set of graphical tools to model object-oriented programs. A *class diagram* visualizes the structure of a software system by showing classes, attributes, and relationships between the classes. Figure 1 shows a slightly simplified UML class diagram of **simFrame**.

In this example, classes are represented by boxes with two parts. The top part contains the name of the class and the bottom part lists its slots. Class names in italics thereby indicate virtual classes. Furthermore, each slot is followed by the name of its class, which can be a basic R data type such as `numeric`, `character`, `logical`, `list` or `function`, but also an S4 class.

Lines or arrows of different forms represent class relationships. Inheritance is denoted by an arrow with an empty triangular head pointing to the superclass. Composition, i.e., a class having another class as a slot, is depicted by an arrow with a solid black diamond on the side of the composed class. A solid line indicates an *association* between two classes. Here an association signals that there is a method with one class as primary input and the other class as output. Last but not least, a dashed line denotes an *association class*, which in the case of **simFrame** is a control class that is not the primary input of the corresponding method but nevertheless determines its behavior.

3.2. Naming conventions

In order to facilitate the usage of the framework, the following naming rules are introduced:

- Names of classes, slots, functions and methods are alphanumeric in mixed case, where the first letter of each internal word is capitalized.
- Class names start with an uppercase letter.
- Functions, methods and slots start with a lowercase letter. Exceptions are functions that initialize a class, which are called *constructors* and have the same name as the class.
- Violate the above rules whenever necessary to maintain compatibility.

These rules are based on code conventions for the programming language Java (e.g., Arnold *et al.* 2005), see `http://java.sun.com/docs/codeconv/`. Some R packages, e.g., **rrcov** (Todorov and Filzmoser 2009; Todorov 2010), use similar rules.

3.3. Accessor and mutator methods

In object-oriented programming languages, *accessor* and *mutator* methods are typically used to retrieve and change the properties of a class, respectively. The idea behind this concept is to hide information about the actual implementation of a class (e.g., what data structures are used) from the user. In **simFrame**, accessors are named `getFoo()` and mutators are named `setFoo()`, where `foo` is the name of the corresponding slot. This naming convention is common in Java and is also used in some R packages (e.g., **rrcov**).

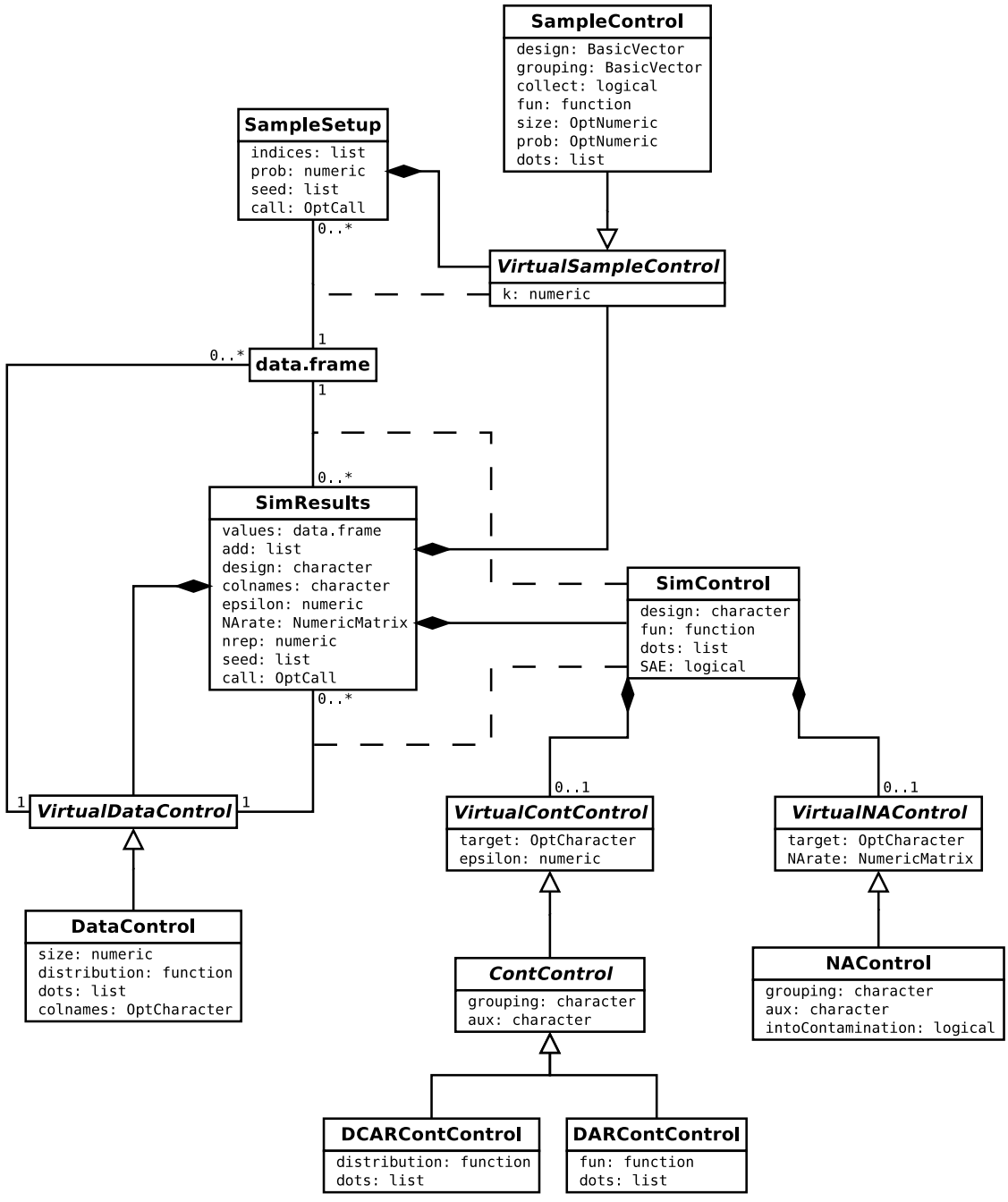

Figure 1: Slightly simplified UML class diagram of **simFrame**.

The use of accessor and mutator methods in **simFrame** is illustrated with the class `NAControl`, which handles the insertion of missing values into a data set (see Section 4.4). Its slot `NArate` controls the proportion of missing values to be inserted.

```
R> nc <- NAControl(NArate = 0.05)
R> getNArate(nc)

[1] 0.05

R> setNArate(nc, c(0.01, 0.03, 0.05, 0.07, 0.09))
R> getNArate(nc)

[1] 0.01 0.03 0.05 0.07 0.09
```

Note that if no method `setFoo()` is available, the slot is not supposed to be changed by the user. However, as already mentioned in Section 2, R allows every slot to be modified with the `@` operator or the `slot()` function.

4. Implementation

The open-source statistical environment R has become the main framework for computing in statistics research. One of its main advantages is that it includes a well-developed programming language and provides interfaces to many others, including the fast low-level languages C and Fortran. The S4 system (Chambers 1998, 2008) complies with all requirements for an object-oriented framework for statistical simulation. Thus most of **simFrame** is implemented as S4 classes and methods, except some utility functions and some C++ code.

Method selection for generic functions is based on *control classes*, which in most cases provides the interfaces for extensions by developers (see Section 7). Most of these generic functions are not expected to be called by the user directly. The idea of the framework is rather to define a number of control objects and to supply them to the function `runSimulation()`, which performs the whole simulation experiment and calls the other functions internally (see Section 4.5 or the examples in Section 6).

4.1. Data handling

In R, data are typically stored in a `data.frame`, and **simFrame** is no exception. However, when samples are taken from a finite population in design-based simulation studies, each observation in the sample represents a number of observations in the population, given by the sample weights. Unless a basic sampling procedure such as simple random sampling is used, the weights are in general not equal for all sampled observations and need to be considered to obtain unbiased estimates. But even if the weights are equal for all observations, they may be needed for the estimation of population totals (e.g., the total turnover of all businesses in a country). In practice, the initial weights are also frequently modified by calibration (e.g., Deville *et al.* 1993), which for simple random sampling is done after post-stratification (e.g., Cochran 1977). Therefore, the sample weights need to be stored.

In addition, the package has been designed with special emphasis on simulations involving typical data problems such as outliers and missing values. It offers mechanisms to contaminate the data and insert missing values so that the influence of these data problems on statistical methods can be investigated, or that outlier detection or imputation methods can be evaluated. The term *contamination* is used in a technical sense here (see Section 4.3 for a definition). Information on which observations are contaminated is often required, both for the user running simulations and for internal use. Since it cannot be retrieved from the data otherwise, it needs to be saved.

As a result, additional variables are added to the data set in these situations. The names of the additional variables are `".weight"` and `".contaminated"`, respectively. Hence these column names should be avoided (which is why they start with a dot), or else the corresponding columns will be overwritten.

Statistical methods often make assumptions about the distribution of the data, e.g., outlier detection methods in multivariate statistics usually assume that the majority of the data follow a multivariate normal distribution. Consequently, such methods are typically tested in simulations on data coming from a certain theoretical distribution. The generation of data from a distributional model is handled by control classes inheriting from the class union (which is a special virtual class with no slots) `VirtualDataControl`. This virtual class is available so that the framework can be extended by the user (see Section 7.1). A simple control class already implemented in **simFrame** is `DataControl`. It consists of the following slots (see also Figure 1):

`size`: The number of observations to be generated.

`distribution`: A function for generating the data, e.g., `rmvnorm` in package **mvtnorm** (Genz and Bretz 2009; Genz *et al.* 2010) for data following a multivariate normal distribution. It should take a positive integer as its first argument (the slot `size` will be passed) and return an object that can be coerced to a `data.frame`.

`dots`: Additional arguments to be passed to `distribution`.

The following example demonstrates how to define a control object for generating data from a multivariate normal distribution.

```
R> library("mvtnorm")
R> dc <- DataControl(size = 10, distribution = rmvnorm, dots =
+     list(mean = rep(0, 2), sigma = matrix(c(1, 0.5, 0.5, 1), 2, 2)))
```

In a model-based simulation study, such a control object is then used by the framework in repeated internal calls of the generic function `generate(control, ...)`.

```
R> foo <- generate(dc)
R> foo

           V1          V2
1  -1.0941320 -0.04443553
```

```
2   0.4403785 -1.98509596
3   0.5454795  0.59987810
4  -0.6966349 -0.67675947
5  -0.7755775 -1.00580145
6  -0.7193342 -1.08787381
7  -0.7331205 -0.13864686
8   0.8982561  0.14180773
9  -0.7294320 -1.01240618
10 -0.1833816  2.11684162
```

While the function `generate()` is designed to be called internally by the simulation framework, it is possible to use it as a general wrapper function for data generation in other contexts. For convenience, the name of the control class may then also be passed to `generate()` as a character string (the default is `"DataControl"`), in which case the slots may be supplied as arguments. Nevertheless, it might be simpler for the user to call the underlying function from the slot `distribution` directly in such applications.

Memory-efficient storage of data frames has recently been added to package **ff** (Adler *et al.* 2010), which might be useful for design-based simulation with large population data. The incorporation into **simFrame** may therefore be investigated as a future task.

4.2. Sampling

A fundamental design principle of **simFrame** in the case of design-based simulation studies is that the sampling procedure is separated from the simulation procedure. Two main advantages arise from *setting up* all samples in advance.

First, the repeated sampling reduces overall computation time dramatically in certain situations, since computer-intensive tasks like stratification need to be performed only once. This is particularly relevant for large population data. As an example, consider the AMELI project that has been mentioned in Section 3. In the close-to-reality simulation studies carried out in this project, up to 10 000 samples are drawn from a population of more than 8 000 000 individuals with stratified sampling or even more complex sampling designs. For such large data sets, stratification takes a considerable amount of time and is a very memory-intensive task. If the samples are taken on-the-fly, i.e., in every simulation run one sample is drawn, the function to take the stratified sample would typically split the population into the different strata in each of the 10 000 iterations. If all samples are drawn in advance, on the other hand, the population data need to be split only once and all 10 000 samples can be taken from the respective strata together.

Second, the samples can be stored permanently, which simplifies the reproduction of simulation results and may help to maximize comparability of results obtained by different partners in a research project. Consider again the AMELI project, where one group of researchers investigates robust semiparametric approaches to improve the estimation of certain indicators (i.e., a distribution is fitted to parts of the data; see Alfons *et al.* 2010e), while another group is focused on nonparametric methods (e.g., trimming or M-estimators; see Hulliger and Schoch 2009b). The aim of this project is to evaluate these methods in realistic settings, therefore the most commonly used sampling designs in real life are applied in the simulation studies. If the two groups use not only the same population data, but also the same previously

set up samples, their results are highly comparable. In addition, the same samples may be used for other close-to-reality simulation studies within the project, e.g., in order to evaluate imputation or outlier detection methods. This is useful in particular for large population data, when complex sampling techniques may be very time-consuming.

In **simFrame**, the generic function `setup(x, control, ...)` is available to set up multiple samples. It returns an object of class `SampleSetup`, which contains the following slots (among others, all slots are shown in Figure 1):

`indices`: A list containing the indices of the sampled observations.

`prob`: A numeric vector giving the inclusion probabilities for every observation of the population. These are necessary to compute the sample weights.

`seed`: A list containing the seeds of the random number generator before and after setting up the samples, respectively.

The function `setup()` may be called by the user to permanently store the samples, but it may also be called internally by the framework if this is not necessary. In any case, methods are selected according to control classes extending `VirtualSampleControl`, which is a virtual class whose only slot `k` specifies the number of samples to be set up. This virtual class provides the interface for extensions by the user (see Section 7.2). The implemented control class `SampleControl` is highly flexible and covers the most frequently used sampling designs in survey statistics:

- Sampling of individual observations with a basic sampling method such as simple random sampling or unequal probability sampling.
- Sampling of whole groups (e.g., households) with a specified sampling method. There are two common approaches towards sampling of groups:
  - Groups are sampled directly. This is usually referred to as *cluster sampling*. However, here the term *cluster* is avoided in the context of sampling to prevent confusion with computer clusters for parallel computing (see Section 5 and the example in Section 6.3).
  - In a first step, individuals are sampled. Then all individuals that belong to the same group as any of the sampled individuals are collected and added to the sample.
- Stratified sampling using one of the above procedures in each stratum.

In addition to the inherited slot `k`, the class `SampleControl` consists of the following slots (see also Figure 1):

`design`: A vector specifying variables to be used for stratification.

`grouping`: A character string, single integer or logical vector specifying a variable to be used for grouping.

`collect`: A logical indicating whether groups should be collected after sampling individuals or sampled directly. The default is to sample groups directly.

`fun`: A function to be used for sampling (the default is simple random sampling). For stratified sampling, this function is applied to each stratum.

`size`: The sample size. For stratified sampling, this should be a numeric vector.

`prob`: A numeric vector giving probability weights.

`dots`: Additional arguments to be passed to `fun`.

Currently, the functions `srs` and `ups` are implemented in **simFrame** for simple random sampling and unequal probability sampling, respectively, but this can easily be extended with user-defined sampling methods (see Section 7.2). Note that the sampling method is evaluated using `try()`. Hence, if an error occurs in obtaining one sample, the others are not lost. This is particularly useful for complex and time-consuming sampling procedures, as the whole process of setting up all samples does not have to be repeated.

The control class for `setup()` may be specified as a character string (the default is, of course, `"SampleControl"`), which allows the slots to be supplied as arguments. Furthermore, `simSample()` is a convenience wrapper for `setup()` with control class `SampleControl`.

To actually draw one of the previously set up samples from the population, the generic function `draw(x, setup, ...)` is used internally by the framework in the simulation runs. It is important to note that the column `".weight"`, which contains the sample weights, is added to the resulting data set. When sampling from finite populations, storing the sample weights is essential. In general, the weights are not equal for all sampled observations, depending on the inclusion probabilities. Hence the sample weights need to be considered in order to obtain unbiased estimates. But even for simple random sampling, when all weights are equal, each observation in the sample represents a number of observations in the population. For the estimation of population totals (e.g., the total turnover of all businesses in a country), the sample weights are thus still necessary. Moreover, the initial sample weights are in practice often modified by calibration (e.g., Deville *et al.* 1993). In the case of simple random sampling, this is done after post-stratification (e.g., Cochran 1977).

In the following illustrative example, two samples from synthetic EU-SILC population data are set up and stored in an object of class `SampleSetup`. EU-SILC is a well-known survey on income and living conditions conducted in European countries (see Section 6.1 for more information and a more detailed example). Afterwards, the first of the two set up samples is drawn from the population.

```
R> data("eusilcP")
R> set <- setup(eusilcP, size = 10, k = 2)
R> summary(set)

2 samples of size 10 are set up

R> set

Indices of observations for each of the 2 samples:

[[1]]
```

```
 [1] 32456 37914 18290 36471 19342 29442 39711 28444 14306 44891

[[2]]
 [1]  4328 18165 42070 29593  8974 29556 28970 44056 10243 49754

R> draw(eusilcP[, c("id", "eqIncome")], set, i = 1)

           id eqIncome .weight
33670 1376802 17760.93  5865.4
10460 1611302 13603.65  5865.4
48757 0782302 13910.35  5865.4
33719 1549903 14641.86  5865.4
55360 0825101 12463.90  5865.4
35123 1251602 27331.19  5865.4
29673 1686001 27981.36  5865.4
26985 1210302  7247.55  5865.4
38182 0611002 19968.32  5865.4
6448  1913501 18091.90  5865.4
```

4.3. Contamination

When evaluating robust statistical methods in simulation studies, a certain part of the data needs to be contaminated, so that the influence of these outliers on the robust estimators (and possibly their classical counterparts) can be studied. The term *contamination* is thereby used in a technical sense in this paper. In robust statistics, the distribution F of contaminated data is typically modeled as a mixture of distributions

$$F = (1 - \varepsilon)G + \varepsilon H, \tag{1}$$

where ε denotes the *contamination level*, G is the distribution of the non-contaminated part of the data and H is the distribution of the contamination (e.g., Maronna *et al.* 2006). Consequently, outliers may be modeled by a two-step process in simulation studies (Béguin and Hulliger 2008; Hulliger and Schoch 2009a):

1. Select the observations to be contaminated. The probabilities of selection may or may not depend on any other information in the data set.
2. Model the distribution of the outliers. The distribution may or may not depend on the original values of the selected observations.

A more detailed mathematical notation of this process with respect to the implementation in **simFrame** can be found in Alfons *et al.* (2010c).

Even though this is a rather simple concept, taking advantage of object-oriented programming techniques such as inheritance allows for a flexible implementation that can be extended by the user with custom contamination models. In **simFrame**, contamination is implemented based on control classes inheriting from `VirtualContControl`. For extensions of the framework, the user may define subclasses of this virtual class (see Section 7.3). Figure 1 displays the full hierarchy of the available control classes for contamination. The basic virtual class contains the following slots:

`target`: A character vector defining the variables to be contaminated, or `NULL` to contaminate all variables (except the additional ones generated internally).

`epsilon`: A numeric vector giving the contamination levels to be used in the simulation.

With the contamination control classes available in **simFrame**, it is possible to contaminate whole groups (e.g., households) rather than individual observations. In addition, the probabilities for selecting items to be contaminated may depend on an auxiliary variable. In order to share these properties, another virtual class called `ContControl` is implemented. These are the additional slots:

`grouping`: A character string specifying a variable to be used for grouping.

`aux`: A character string specifying an auxiliary variable whose values are used as probability weights for selecting the items (observations or groups) to be contaminated.

The distribution of the contaminated data in simulation experiments may or may not depend on the original values. Similar to model-based data generation (see Section 4.1), the control class `DCARContControl` supports specifying a distribution function for generating the contamination. DCAR stands for distributed completely at random and corresponds to contamination independent of the original data. If a variable for grouping is specified, the same values are used for all observations in the same group. `DCARContControl` extends `ContControl` by the following slots:

`distribution`: A function for generating the data for the contamination, e.g., `rmvnorm` in package **mvtnorm** for a multivariate normal distribution.

`dots`: Additional arguments to be passed to `distribution`.

On the other hand, contamination based on the original values is realized by the control class `DARContControl`. DAR thereby stands for distributed at random. An arbitrary function may be used to modify the data. To do so, the original values of the observations to be contaminated are passed as its first argument. Thus the following slots are available in addition to those from `ContControl`:

`fun`: A function generating the values of the contaminated data based on the original values.

`dots`: Additional arguments to be passed to `fun`.

In the following example, a control object of class `DARContControl` is defined. The contamination level is set to 20% and the specified function multiplies the original values from variable `"V2"` of the observations to be contaminated by a factor 100.

```
R> cc <- DARContControl(target = "V2", epsilon = 0.2,
+     fun = function(x) x * 100)
```

If a control object for contamination is supplied, the framework calls the generic function `contaminate(x, control, ...)` in the simulation runs internally to add the contamination. In many applications, it is necessary to know which observations were contaminated, e.g., to evaluate outlier detection methods. Hence a logical variable, which is called `".contaminated"` and indicates the contaminated observations, is added to the resulting data set. As an example, the data generated in Section 4.1 is contaminated below.

```
R> bar <- contaminate(foo, cc)
R> bar
```

```
           V1           V2 .contaminated
1  -1.0941320   -4.4435535          TRUE
2   0.4403785   -1.9850960         FALSE
3   0.5454795    0.5998781         FALSE
4  -0.6966349   -0.6767595         FALSE
5  -0.7755775   -1.0058015         FALSE
6  -0.7193342   -1.0878738         FALSE
7  -0.7331205   -0.1386469         FALSE
8   0.8982561    0.1418077         FALSE
9  -0.7294320 -101.2406178          TRUE
10 -0.1833816    2.1168416         FALSE
```

Despite being designed for internal use in the simulation procedure, `contaminate()` also allows the control class to be specified as a character string (with `"DCARContControl"` being the default). In this case the slots may be supplied as arguments.

4.4. Insertion of missing values

Missing values are included in many data sets, in particular survey data hardly ever contain complete information. In practice, missing values often need to be imputed, which results in additional uncertainty in further statistical analysis (e.g., Little and Rubin 2002). This additional variability needs to be considered when computing variance estimates or confidence intervals. In simulation studies, it may therefore be of interest to study the properties of different imputation methods or to investigate the influence of missing values on point and variance estimates.

Three mechanisms generating missing values are commonly distinguished in the literature addressing missing data (e.g., Little and Rubin 2002):

- Missing completely at random (MCAR): The probability of missingness does not depend on any observed or missing information.
- Missing at random (MAR): The probability of missingness depends on the observed information.
- Missing not at random (MNAR): The probability of missingness depends on the missing information itself and may also depend on the observed information.

Similar to the implementation of the functionality for contamination, the insertion of missing data is handled by control classes extending `VirtualNAControl` (the hierarchy of the control classes is shown in Figure 1). This virtual class is the basis for extensions by the user (see Section 7.4). It consists of the following slots:

`target`: A character vector specifying the variables into which missing values should be inserted, or `NULL` to insert missing values into all variables (except the additional ones generated internally).

`NArate`: A numeric vector or matrix giving the missing value rates to be used in the simulation.

It should be noted that missing value rates may be selected individually for the target variables. The same missing value rates are used for all target variables if they are specified as a vector. If a matrix is supplied, on the other hand, the missing value rates to be used for each target variable are given by the respective column.

Extending `VirtualNAControl`, the control class `NAControl` allows whole groups to be set as missing rather than individual values. To account for MAR or MNAR situations instead of MCAR, an auxiliary variable of probability weights may be specified for each target variable. Furthermore, when studying robust methods for the analysis or imputation of incomplete data, it is sometimes desired to insert missing values only into non-contaminated observations. In other situations, a more realistic scenario in which missing values are also inserted into contaminated observations may be preferred. Both scenarios are implemented in the framework. These are the additional slots of `NAControl`:

`grouping`: A character string specifying a variable to be used for grouping.

`aux`: A character vector specifying auxiliary variables whose values are used as probability weights for selecting the values to be set as missing in the respective target variables.

`intoContamination`: A logical indicating whether missing values should also be inserted into contaminated observations. The default is to insert missings only into non-contaminated observations.

The following example shows how to define a control object of class `NAControl` that corresponds to an MCAR situation. For all variables, 30% of the values will be set as missing. However, missing values will only be inserted into non-contaminated observations.

```
R> nc <- NAControl(NArate = 0.3)
```

If a control object for missing data is supplied, the generic function `setNA(x, control, ...)` is called internally by the framework in the simulation runs to set the missing values. Below, missing values are inserted into the contaminated data from the previous section.

```
R> setNA(bar, nc)
```

```
           V1           V2 .contaminated
1  -1.0941320   -4.4435535          TRUE
2          NA   -1.9850960         FALSE
3          NA    0.5998781         FALSE
4          NA           NA         FALSE
5  -0.7755775           NA         FALSE
6  -0.7193342   -1.0878738         FALSE
7  -0.7331205           NA         FALSE
8   0.8982561    0.1418077         FALSE
9  -0.7294320 -101.2406178          TRUE
10 -0.1833816    2.1168416         FALSE
```

As `contaminate()`, the function `setNA()` is designed for internal use in the simulation procedure. Nevertheless, it is possible to supply the name of the control class as a character string (the default is `"NAControl"`), which allows the slots to be supplied as arguments.

4.5. Running simulations

The central component of the simulation framework is the generic function `runSimulation()`, which combines all the elements of the package into one convenient interface for running simulation studies. Based on a collection of control objects, it allows to perform even complex simulation experiments with just a few lines of code. Switching between simulation designs is possible with minimal programming effort as well, only some control objects need to be defined or modified. For design-based simulation, population data and a control object for sampling or previously set up samples may be passed to `runSimulation()`. For model-based simulation, on the other hand, a control object for data generation and the number of replications may be supplied.

In addition, the control class `SimControl` determines how the simulation runs are performed. These are the slots of `SimControl` (see also Figure 1):

`contControl`: A control object for contamination.

`NAControl`: A control object for inserting missing values.

`design`: A character vector specifying variables to be used for splitting the data into domains and performing the simulations on every domain.

`fun`: The function to be applied in the simulation runs.

`dots`: Additional arguments to be passed to `fun`.

`SAE`: A logical indicating whether small area estimation (see, e.g., Rao 2003) will be used in the simulation.

Most importantly, the function to be applied in the simulation runs needs to be defined. There are some requirements for the function:

- It must return a numeric vector, or a list with the two components `values` (a numeric vector) and `add` (additional results of any class, e.g., statistical models). Note that the latter is computationally slightly more expensive.

- A `data.frame` is passed to `fun` in every simulation run. The corresponding argument must be called `x`.
- If comparisons with the original data need to be made, e.g., for evaluating the quality of imputation methods, the function should have an argument called `orig`.
- If different domains are used in the simulation, the indices of the current domain can be passed to the function via an argument called `domain`.

One of the most important features of **simFrame** is that the supplied function is evaluated using `try()`. Therefore, if computations fail in one of the simulation runs, `runSimulation()` simply continues with the next run. The results from previous runs are not lost and the computation time has not been spent in vain.

Furthermore, control classes for adding contamination and missing values may be specified. In design-based simulations, contamination and nonresponse are added to the samples rather than the population, for maximum control over the amount of outliers or missing values (cf. Alfons *et al.* 2009). Another useful feature is that the data may be split into different domains. The simulations, including contamination and the insertion of missing values, are then performed on every domain separately, unless small area estimation is used.

Concerning small area estimation, the following points have to be kept in mind. The design for splitting the data must be supplied and `SAE` must be set to `TRUE`. However, the data are not actually split into the specified domains. Instead, the whole data set is passed to the specified function. Also contamination and missing values are added to the whole data. Last, but not least, the function for the simulation runs must have a `domain` argument so that the current domain can be extracted from the whole data. In any case, small area estimation is not a main focus of the current version of **simFrame** and will therefore not be discussed further in this paper. Improving the support for small area estimation is future work.

For user convenience, the slots of the `SimControl` object may also be supplied as arguments. After running the simulations, the results of the individual simulation runs are combined and packed into an object of class `SimResults`. The most important slots are (see Figure 1 for a complete list):

`values`: A `data.frame` containing the simulation results.

`add`: A list containing additional simulation results, e.g., statistical models.

`epsilon`: The contamination levels used in the simulation.

`NArate`: The missing value rates used in the simulation.

`seed`: A list containing the seeds of the random number generator before and after the simulation, respectively.

An illustrative example for the use of `runSimulation()` is given in the following design-based simulation experiment. The synthetic EU-SILC example data of the package is thereby used as population data. It contains information about household income, but data is also available on the personal level (see Section 6.1 for more information on the data). From this data set, 50 samples of 500 persons are drawn with simple random sampling. In addition, the equivalized

income of 2% of the sampled persons are multiplied by a factor 25. In every simulation run, the population mean income is estimated with the mean and the 2% trimmed mean of the sample. With the following commands, control objects for sampling and contamination are defined, along with the function for the simulation. Before calling `runSimulation()`, the seed of the random number generator is set for reproducibility of the results.

```
R> data("eusilcP")
R> sc <- SampleControl(size = 500, k = 50)
R> cc <- DARContControl(target = "eqIncome", epsilon = 0.02,
+     fun = function(x) x * 25)
R> sim <- function(x) {
+     c(mean = mean(x$eqIncome), trimmed = mean(x$eqIncome, trim = 0.02))
+ }
R> set.seed(12345)
R> results <- runSimulation(eusilcP, sc, contControl = cc, fun = sim)
```

Methods for several frequently used generic functions are available to inspect the simulation results. Besides `head()`, `tail()` and `summary()` methods, a method for `aggregate()` is implemented. The latter can be used to calculate summary statistics of the results. By default, the mean is used as summary statistic. Depending on the simulation design, the summary statistics are are computed for different subsets of the results. These subsets are thereby given by the different combinations of contamination levels (if contamination is used), missing value rates (if missing values are inserted) and domains (if the simulations are performed on different domains of the data).

Below, the first parts of the simulation results are returned using `head()` and the average results are computed with `aggregate()`. For comparison, the true population mean is computed afterwards.

```
R> head(results)

  Run Sample Epsilon     mean  trimmed
1   1      1    0.02 29609.10 20737.05
2   2      2    0.02 28834.87 20023.74
3   3      3    0.02 29819.41 20410.12
4   4      4    0.02 28840.16 20438.05
5   5      5    0.02 27323.11 19298.89
6   6      6    0.02 29614.79 20442.76

R> aggregate(results)

  Epsilon     mean  trimmed
1    0.02 29697.64 20361.22

R> tv <- mean(eusilcP$eqIncome)
R> tv

[1] 20162.8
```

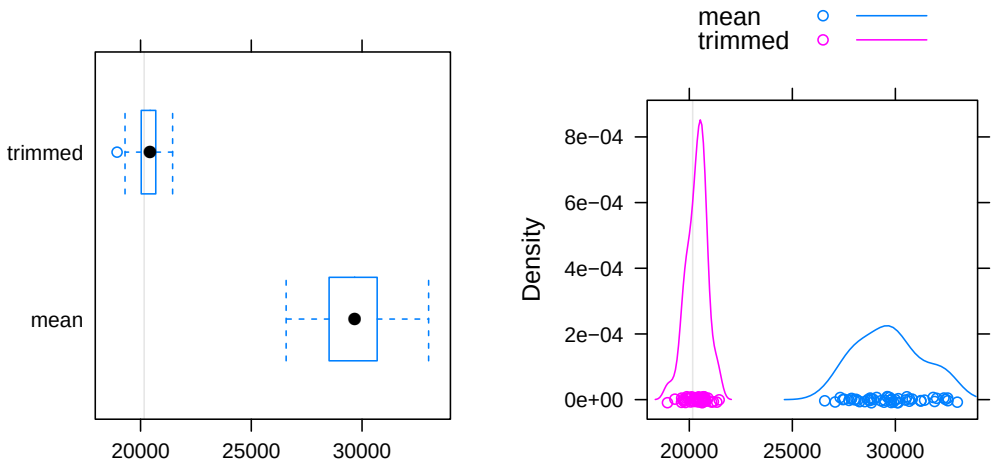

Figure 2: Simulation results from the simple illustrative example. *Left*: Default plot of results from a simulation study with one contamination level, in this example obtained by `plot(results, true = tv)`. *Right*: Kernel density plots of the simulation results, obtained by `simDensityplot(results, true = tv)`.

Various plots for simulation results are implemented in the framework, as discussed in the following section. In Figure 2, the results for this illustrative example are displayed by box plots and kernel density plots. The plots show the well-known fact that the mean is highly influenced by outliers. While the trimmed mean is not influenced by the contamination and has much smaller variance, there is still some bias. Since the outliers in this example are only in the upper tail of the data, the remaining bias results from trimming lower part as well.

Section 6 contains more elaborate examples for design-based and model-based simulation with detailed step-by-step instructions, as well as some motivation and interpretation.

4.6. Visualization

Visualization methods for the simulation results are based on **lattice** graphics (Sarkar 2008, 2010). If the simulation study has been divided into several domains, the results for each domain are displayed in a separate panel. Box plots and kernel density plots are implemented in the functions `simBwplot()` and `simDensityplot()`, respectively. For simulations involving different contamination levels or missing value rates, `simXyplot()` plots the average results against the contamination levels or missing value rates. In all of these plots, reference lines for the true values can be added. Moreover, the `plot()` method for the class `SimResults` selects a suitable graphical representation of the simulation results automatically.

Figure 2 shows the default plot and kernel density plots for the simulation results from the simple illustrative example in the previous section. Further examples for the visualization of simulation results are given in Section 6.

5. Parallel computing

Statistical simulation is *embarrassingly parallel*, hence computational performance can be increased by parallel computing. Since version 0.5.0, parallel computing in **simFrame** is implemented using the package **parallel**, which is part of the R base distribution since version 2.14.0 and builds upon work done for the contributed packages **multicore** (Urbanek 2009) and **snow** (Rossini *et al.* 2007; Tierney *et al.* 2008, 2009). The latter was recommended by Schmidberger *et al.* (2009) in an analysis of the state-of-the-art in parallel computing with R. For setting up multiple samples and running simulations on a cluster, the functions `clusterSetup()` and `clusterRunSimulation()` are implemented. Note that all objects and packages required for the computations (including **simFrame**) need to be made available on every worker process unless the worker processes are created by forking. An example for parallel computing is presented in Section 6.3.

In order to ensure reproducibility of the simulation results, random number streams should be used. For this purpose, **parallel** contains an R re-implementation of the C library **RngStreams** (L'Ecuyer *et al.* 2002). Random number streams can thereby be created via the function `clusterSetRNGStream()`. It should be noted that the packages **rlecuyer** (Sevcikova and Rossini 2009) and **rstream** (L'Ecuyer and Leydold 2005; Leydold 2010) provide interfaces to the C library by L'Ecuyer *et al.* (2002).

6. Using the framework

In this section, the use of **simFrame** is demonstrated on examples for design-based and model-based simulation. An example for parallel computing is included as well. Note that the only purpose of these examples is to illustrate the use of the package. It is not the aim of this section to provide a thorough analysis of the presented methodology, as this is beyond the scope of this paper.

6.1. Design-based simulation

The Laeken indicators are a set of indicators used to measure social cohesion in member states of the European Union and other European countries (cf. Atkinson *et al.* 2002). Most of the Laeken indicators are computed from EU-SILC (European Union Statistics on Income and Living Conditions) survey data. Synthetic EU-SILC data based on the Austrian sample from 2006 is included in **simFrame**. This data set was generated using the synthetic data generation framework by Alfons *et al.* (2010b) from package **simPopulation** (Alfons and Kraft 2010). It consists of 25 000 households with data available on the personal level, and is used as population data in this example. Note that this is an illustrative example, as the data set does not represent the true population sizes of Austria and its regions.

While only being a secondary Laeken indicator, the *Gini coefficient* is a frequently used measure of inequality and is widely studied in the literature. In the case of EU-SILC, the Gini coefficient is calculated based on an equivalized household income. In this example, the standard estimation method (EU-SILC 2004) is compared to two semiparametric approaches, which fit a Pareto distribution (e.g., Kleiber and Kotz 2003) to the upper tail of the data. Hill (1975) introduced the maximum-likelihood estimator, which is thus referred to as Hill estimator. The partial density component (PDC) estimator (Vandewalle *et al.* 2007), on the

other hand, follows a robust approach. These methods are available in the R package **laeken** (Alfons *et al.* 2010a). A more detailed discussion on Pareto tail modeling with application to selected Laeken indicators can be found in Alfons *et al.* (2010e).

First, the required package and the data set need to be loaded. Furthermore, the seed of the random number generator is set for reproducibility of the results.

```
R> library("laeken")
R> data("eusilcP")
R> set.seed(12345)
```

Next, 100 samples of 1500 households are set up. Stratified sampling by regions combined with sampling of whole households rather than individuals can be achieved with one command.

```
R> set <- setup(eusilcP, design = "region", grouping = "hid",
+     size = c(75, 250, 250, 125, 200, 225, 125, 150, 100), k = 100)
```

Since a robust method is going to be compared to two classical ones, a control object for contamination is defined. EU-SILC data typically contain a very low amount of outliers, therefore the equivalized household income of 0.5% of the households is contaminated. In addition, the contamination is generated by a normal distribution $\mathcal{N}(\mu, \sigma^2)$ with mean $\mu = 500\,000$ and standard deviation $\sigma = 10\,000$.

```
R> cc <- DCARContControl(target = "eqIncome", epsilon = 0.005,
+     grouping = "hid", dots = list(mean = 500000, sd = 10000))
```

The function for the simulation runs is quite simple as well. Its argument `k` determines the number of households whose income is modeled by a Pareto distribution.

```
R> sim <- function(x, k) {
+     g <- gini(x$eqIncome, x$.weight)$value
+     eqIncHill <- fitPareto(x$eqIncome, k = k,
+         method = "thetaHill", groups = x$hid)
+     gHill <- gini(eqIncHill, x$.weight)$value
+     eqIncPDC <- fitPareto(x$eqIncome, k = k,
+         method = "thetaPDC", groups = x$hid)
+     gPDC <- gini(eqIncPDC, x$.weight)$value
+     c(standard = g, Hill = gHill, PDC = gPDC)
+ }
```

With all necessary objects available, running the simulation experiment is only one more command. Note that simulations are performed separately for each gender. The value of `k` for the Pareto distribution is thereby set to 125.

```
R> results <- runSimulation(eusilcP, set, contControl = cc,
+     design = "gender", fun = sim, k = 125)
```

The `head()` and `aggregate()` methods are used to take a look at the simulation results. In this case, `aggregate()` computes the average results for each subset.

```
R> head(results)
```

```
  Run Sample Epsilon gender standard     Hill      PDC
1   1      1   0.005   male 34.58446 29.96658 26.61415
2   1      1   0.005 female 38.82356 33.93700 28.82045
3   2      2   0.005   male 34.34853 29.09325 27.66380
4   2      2   0.005 female 36.38429 30.06097 27.42663
5   3      3   0.005   male 33.39992 30.54211 23.96698
6   3      3   0.005 female 35.12883 30.51336 26.06518
```

```
R> aggregate(results)
```

```
  Epsilon gender standard     Hill      PDC
1   0.005   male 33.18580 29.00265 26.21119
2   0.005 female 35.61341 31.28984 27.69054
```

For comparison with the simulation results, the true values of the Gini coefficient need to be computed. These can be added as reference lines to the plot of the simulation results (see Figure 3).

```
R> tv <- simSapply(eusilcP, "gender", function(x) gini(x$eqIncome)$value)
```

Figure 3 shows that even a small proportion of outliers completely corrupts the standard estimation of the Gini coefficient. Also fitting the Pareto distribution with the Hill estimator is highly influenced by contamination, whereas the robust PDC estimator leads to excellent results. But most importantly, this example shows that even complex simulation designs require only a few lines of code.

Further examples for design-based simulation that demonstrate the strengths of the framework can be found in a supplementary paper. This supplementary paper is also included in **simFrame** as a package vignette (Leisch 2003).

6.2. Model-based simulation

In this section, model-based simulation is demonstrated using an example for compositional data. An observation $\boldsymbol{x} = (x_1, \ldots, x_D)$ is by definition a *D-part composition* if, and only if, $x_i > 0$, $i = 1, \ldots, D$, and all relevant information is contained in the ratios between the components (Aitchison 1986). Consequently, compositional data contain only relative information. The information is essentially the same if an observation is multiplied with a positive constant. But if the value of one component changes, the other components need to change accordingly. Examples for compositional data are element concentrations in chemical analysis of a sample material or monthly household expenditures on different spending categories such as housing, food or leisure activities.

It is important to note that compositional data have no direct representation in the Euclidean space and that their geometry is entirely different (see Aitchison 1986). The sample space of

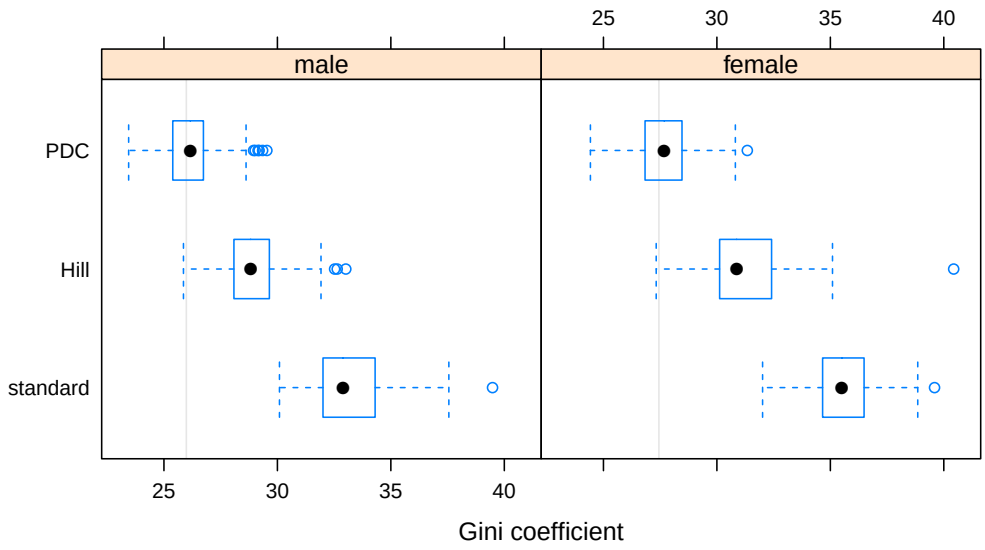

Figure 3: Default plot of results from a simulation study with one contamination level and different domains, in this example obtained by `plot(results, true = tv, xlab = "Gini coefficient")`.

D-part compositions is called the *simplex* and a suitable distance measure is the *Aitchison distance* d_{A} (Aitchison 1992; Aitchison *et al.* 2000). Fortunately, there exists an isometric transformation from the D-dimensional simplex to $\mathbb{R}^{D-1}$, which is called the *isometric logratio* (ilr) transformation (Egozcue *et al.* 2003). With this transformation, the Aitchison distance can be expressed as

$$d_{\mathrm{A}}(\boldsymbol{x}, \boldsymbol{y}) = d_{\mathrm{E}}(\mathrm{ilr}(\boldsymbol{x}), \mathrm{ilr}(\boldsymbol{y})), \tag{2}$$

where d_{E} denotes the Euclidean distance.

Hron *et al.* (2010) introduced imputation methods for compositional data, which are implemented in the R package **robCompositions** (Templ *et al.* 2009, 2010). While the package is focused on robust methods, only classical imputation methods are used in this example. The first method is a modification of k-nearest neighbor ($k$nn) imputation (Troyanskaya *et al.* 2001), the second follows an iterative model-based approach using least squares (LS) regression.

Before any computations are performed, the required packages are loaded and the seed of the random number generator is set for reproducibility.

```
R> library("robCompositions")
R> library("mvtnorm")
R> set.seed(12345)
```

The data in this example are generated by a *normal distribution on the simplex*, denoted by $\mathcal{N}_{\mathcal{S}}^{D}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ (e.g., Mateu-Figueras *et al.* 2008). A random composition $\boldsymbol{x} = (x_1, \ldots, x_D)$ follows this distribution if, and only if, the vector of ilr transformed variables follows a multivariate

normal distribution on $\mathbb{R}^{D-1}$ with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$. The following commands create a control object for generating 150 realizations of a random variable $\boldsymbol{X} \sim \mathcal{N}_{\mathcal{S}}^4(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with

$$\boldsymbol{\mu} = \begin{pmatrix} 0 \\ 2 \\ 3 \end{pmatrix} \quad \text{and} \quad \boldsymbol{\Sigma} = \begin{pmatrix} 1 & -0.5 & 1.4 \\ -0.5 & 1 & -0.6 \\ 1.4 & -0.6 & 2 \end{pmatrix}.$$

```
R> crnorm <- function(n, mean, sigma) isomLRinv(rmvnorm(n, mean, sigma))
R> sigma <- matrix(c(1, -0.5, 1.4, -0.5, 1, -0.6, 1.4, -0.6, 2), 3, 3)
R> dc <- DataControl(size = 150, distribution = crnorm,
+     dots = list(mean = c(0, 2, 3), sigma = sigma))
```

Furthermore, a control object for inserting missing values needs to be created. In every variable, 5% of the observations are set as missing completely at random.

```
R> nc <- NAControl(NArate = 0.05)
```

For the two selected imputation methods, the *relative Aitchison distance* between the original and the imputed data (cf. the simulation study in Hron *et al.* 2010) is computed in every simulation run.

```
R> sim <- function(x, orig) {
+     i <- apply(x, 1, function(x) any(is.na(x)))
+     ni <- length(which(i))
+     xKNNa <- impKNNa(x)$xImp
+     xLS <- impCoda(x, method = "lm")$xImp
+     c(knn = aDist(xKNNa, orig)/ni, LS = aDist(xLS, orig)/ni)
+ }
```

The simulation can then be run with the following command:

```
R> results <- runSimulation(dc, nrep = 50, NAControl = nc, fun = sim)
```

As in the previous example, the results are inspected using `head()` and `aggregate()`.

```
R> head(results)
```

```
  Run Rep NArate       knn        LS
1   1   1   0.05 0.3331367 0.2446087
2   2   2   0.05 0.3473012 0.1723409
3   3   3   0.05 0.3785610 0.2204749
4   4   4   0.05 0.5671431 0.3205881
5   5   5   0.05 0.4471062 0.3002530
6   6   6   0.05 0.4615397 0.3571398
```

```
R> aggregate(results)
```

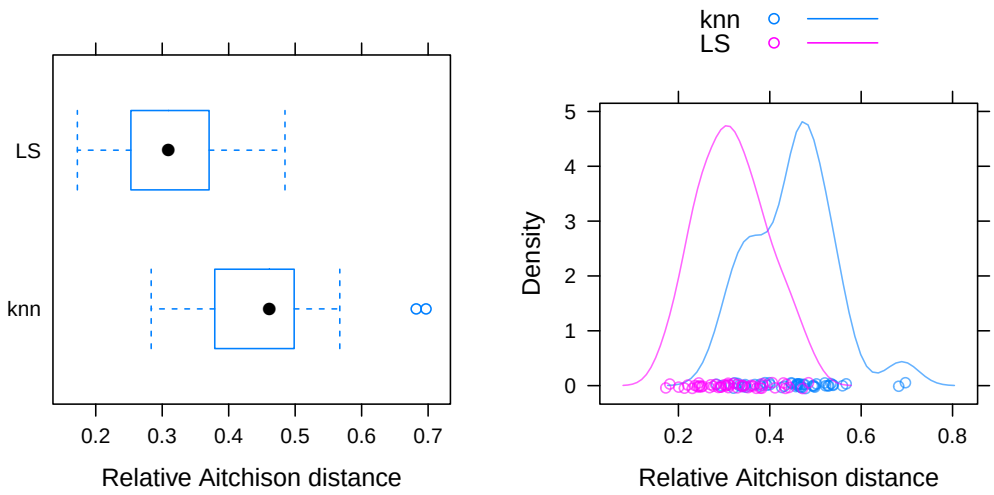

Figure 4: *Left*: Default plot of results from a simulation study with one missing value rate, in this example obtained by `plot(results, xlab = "Relative Aitchison distance")`. *Right*: Kernel density plots of the simulation results, obtained by `simDensityplot(results, alpha = 0.6, xlab = "Relative Aitchison distance")`.

```
  NArate       knn        LS
1   0.05 0.4496368 0.3181275
```

Box plots and kernel density plots of the simulation results are presented in Figure 4. Since the imputation methods in this example are evaluated in terms of a relative distance measure, values closer to 0 indicate better performance. Clearly, the iterative model-based procedure leads to better results than the modified $k$nn approach with respect to the relative Aitchison distance. This is not a surprising result, as the latter is used as a starting point in the iterative procedure. For serious evaluation of the imputation methods, however, also other criteria need to be taken into account (e.g., how well the variability of the multivariate data is reflected; see Hron *et al.* 2010).

6.3. Parallel computing

Using parallel computing, computation time may be significantly decreased in statistical simulation. In this section, the example for model-based simulation from before is extended to more than one missing value rate. Hence some of the objects are already defined above, but in order to provide a complete description on how to perform parallel computing with **simFrame**, these definitions are repeated here.

The first step is to start a cluster for parallel computing. In this example, four parallel worker processes on the local machine are initialized.

```
R> cl <- makeCluster(4, type="PSOCK")
```

All the functions and packages required for the computations (including **simFrame**) need to be loaded on the worker processes.

```
R> clusterEvalQ(cl, {
+         library("simFrame")
+         library("robCompositions")
+         library("mvtnorm")
+     })
```

For reproducibility of the results, a random number stream is generated.

```
R> clusterSetRNGStream(cl, iseed=12345)
```

Control objects for data generation and the insertion of missing values, as well as the function for the simulation runs are defined as in the previous section. The only difference is that multiple missing value rates (1%, 3%, 5%, 7% and 9%) are used in this example.

```
R> crnorm <- function(n, mean, sigma) isomLRinv(rmvnorm(n, mean, sigma))
R> sigma <- matrix(c(1, -0.5, 1.4, -0.5, 1, -0.6, 1.4, -0.6, 2), 3, 3)
R> dc <- DataControl(size = 150, distribution = crnorm,
+     dots = list(mean = c(0, 2, 3), sigma = sigma))
R> nc <- NAControl(NArate = c(0.01, 0.03, 0.05, 0.07, 0.09))
R> sim <- function(x, orig) {
+     i <- apply(x, 1, function(x) any(is.na(x)))
+     ni <- length(which(i))
+     xKNNa <- impKNNa(x)$xImp
+     xLS <- impCoda(x, method = "lm")$xImp
+     c(knn = aDist(xKNNa, orig)/ni, LS = aDist(xLS, orig)/ni)
+ }
```

These objects need to be made available on the worker processes. Since they are small in size, they are exported. Note that large objects, e.g., data sets for design-based simulation, should rather be constructed on the worker processes, as computation is much faster than network communication (Schmidberger *et al.* 2009).

```
R> clusterExport(cl, c("crnorm", "sigma", "dc", "nc", "sim"))
```

Then only one more command is needed to run the simulation.

```
R> results <- clusterRunSimulation(cl, dc, nrep = 50, NAControl = nc, fun = sim)
```

Last, the cluster needs to be stopped after carrying out the simulation study in order to ensure that the worker processes are properly shut down.

```
R> stopCluster(cl)
```

After the parallel computations have finished, the simulation results can be inspected as usual. In this example, the `aggregate()` method returns the average results of the relative distances for each missing value rate.

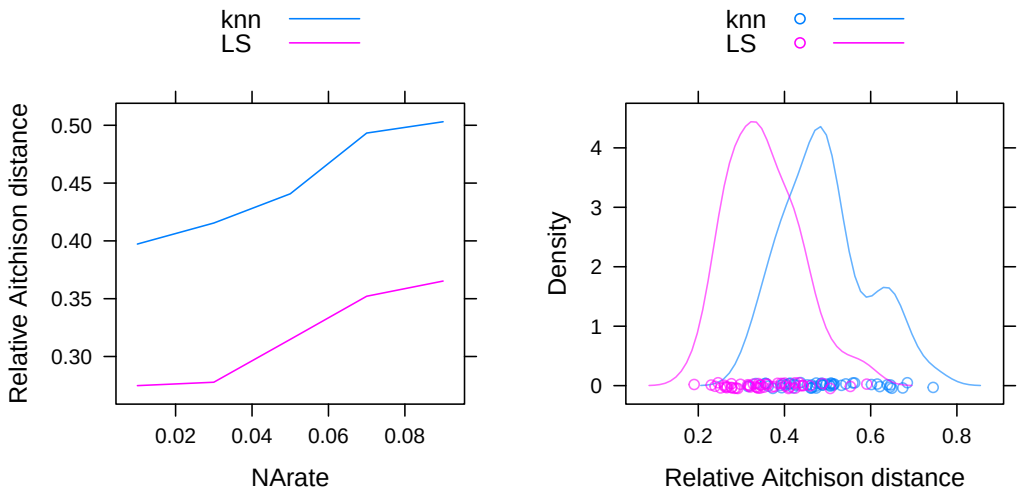

Figure 5: *Left*: Default plot of results from a simulation study with multiple missing value rates, in this example obtained by `plot(results, ylab = "Relative Aitchison distance")`. *Right*: Kernel density plots of the simulation results for a specified missing value rate (7%), obtained by `simDensityplot(results, NArate=0.07, alpha = 0.6, xlab = "Relative Aitchison distance")`.

```
R> head(results)
```

```
  Run Rep NArate       knn        LS
1   1   1   0.01 0.3567153 0.2094683
2   2   1   0.03 0.5885490 0.1892476
3   3   1   0.05 0.3640092 0.3282235
4   4   1   0.07 0.3624546 0.3359033
5   5   1   0.09 0.4182588 0.3362382
6   6   2   0.01 0.2154518 0.1934485
```

```
R> aggregate(results)
```

```
  NArate       knn        LS
1   0.01 0.3972597 0.2747889
2   0.03 0.4154468 0.2777312
3   0.05 0.4407218 0.3147971
4   0.07 0.4932648 0.3521267
5   0.09 0.5030542 0.3652661
```

Figure 5 visualizes the simulation results. On the left hand side, the average relative Aitchison distances are plotted against the missing value rates. On the right hand side, kernel density plots for a specified missing value rate (7%) is shown. The results are not much different from those in the previous section. Note that the difference of the average results for the two methods remains quite constant in this simulation example.

7. Extending the framework

One of the main advantages of the S4 implementation of **simFrame** is that it provides clear interfaces for user-defined extensions. With the available control classes for data generation, sampling, contamination and the insertion of missing data, the framework is highly flexible and can be used for a wide range of simulation designs. Nevertheless, extensions may sometimes be desired for specialized functionality. In order to extend the framework, developers can implement custom control classes and the corresponding methods.

7.1. Model-based data

The control class `DataControl` available in **simFrame** is quite simple but general. For user-defined data generation models, it often suffices to implement a function and use it as the `distribution` slot in the `DataControl` object. This function should have the number of observations to be generated as its first argument, as illustrated in the code skeleton in Figure 6 (*top*). The name of the argument is thereby not important. Furthermore, the function should return an object that can be coerced to a `data.frame`.

However, if more specialized data generation models are required, the framework can be extended by defining a control class extending `VirtualDataControl` and the corresponding method for the generic function `generate()`. If, e.g., a specific distribution or mixture of distributions is frequently used in simulation experiments, a distinct control class may be more convenient for the user. Figure 6 (*bottom*) contains the code skeleton for such an extension.

7.2. Sampling

In **simFrame**, the control class `SampleControl` is highly flexible and allows stratified sampling as well as sampling of whole groups rather than individuals with a specified sampling method. Hence it is often sufficient to implement the desired sampling method for the simple non-stratified case to extend the existing framework. However, there are some restrictions on the

```
myDataGeneration <- function(size, ...) {
    # computations
}
```

```
setClass("MyDataControl",
    # class definition
    contains = "VirtualDataControl")

setMethod("generate",
    signature(control = "MyDataControl"),
    function(control) {
        # method definition
    })
```

Figure 6: *Top:* Code skeleton for a user-defined data generation method. *Bottom:* Code skeleton for extending model-based data generation with a custom control class and the corresponding method for `generate()`.

argument names of the function, which should return a vector containing the indices of the sampled observations.

- If the sampling method needs population data as input, the corresponding argument should be called `x` and should expect a `data.frame`.
- If it only needs the population size as input, the argument should be called `N`.
- If necessary, the argument for the sample size should be called `size`.
- If necessary, the argument for the probability weights should be called `prob`.

Note that the function is not expected to have both `x` and `N` as arguments, and that the latter is much faster for stratified sampling or group sampling. Furthermore, a function with `prob` as its only argument is perfectly valid (for probability proportional to size sampling). Figure 7 (*top*) shows an example for Poisson sampling using the implementation in package **sampling** (Tillé and Matei 2009).

Nevertheless, for very complex sampling procedures, it is possible to define a control class extending `VirtualSampleControl` and the corresponding `setup()` method. The code skeleton for such an extension is shown in Figure 7 (*bottom*). In order to optimize computational performance, it is necessary to efficiently set up multiple samples. Thereby the slot `k` of `VirtualSampleControl` needs to be used to control the number of samples, and the resulting object must be of class `SampleSetup`. For using parallel computing to set up samples with a self-defined control class, a method for `clusterSetup()` may be defined.

7.3. Contamination

A wide range of contamination models is covered by the control classes `DCARContControl` and `DARContControl`. However, other contamination models can be added by defining a control class inheriting from `VirtualContControl` and the corresponding method for `contaminate()` (see the code skeleton in Figure 8). Note that `VirtualContControl` contains the slots `target` and `epsilon` for selecting the target variable(s) and contamination level(s), respectively. In case the contaminated observations need to be identified at a later stage of the simulation, e.g., if conflicts with inserting missing values should be avoided, a logical indicator variable `".contaminated"` should be added to the returned data set.

7.4. Insertion of missing values

Similar to extending the framework for model-based data generation and contamination, user-defined missing value models can be added by defining a control class extending the virtual class `VirtualNAControl` and the corresponding method for the generic function `setNA()` (see the code skeleton in Figure 9). The slots `target` and `NArate` for selecting the target variable(s) and missing value rate(s), respectively, are inherited from `VirtualNAControl`.

8. Conclusions and outlook

The flexible, object-oriented implementation of **simFrame** allows researchers to make use of a wide range of simulation designs with a minimal effort of programming. Control classes

```
myPoisson <- function(prob) {
    require(sampling)
    which(as.logical(UPpoisson(prob)))
}
```

```
setClass("MySampleControl",
    # definition of additional properties
    contains = "VirtualSampleControl")

setMethod("setup",
    signature(x = "data.frame", control = "MySampleControl"),
    function(x, control) {
        # method definition
    })

setMethod("clusterSetup",
    signature(x = "data.frame", control = "MySampleControl"),
    function(cl, x, control) {
        # method definition
    })
```

Figure 7: *Top:* User-defined function for Poisson sampling. *Bottom:* Code skeleton for user-defined setup of multiple samples with a custom control class and the corresponding methods for `setup()` and `clusterSetup()`.

```
setClass("MyContControl",
    # definition of additional properties
    contains = "VirtualContControl")

setMethod("contaminate",
    signature(x = "data.frame", control = "MyContControl"),
    function(x, control, i) {
        # method definition
    })
```

Figure 8: Code skeleton for a user-defined control class for contamination and the corresponding method for `contamintate()`.

```
setClass("MyNAControl",
    # definition of additional properties
    contains = "VirtualNAControl")

setMethod("setNA",
    signature(x = "data.frame", control = "MyNAControl"),
    function(x, control, i) {
        # method definition
    })
```

Figure 9: Code skeleton for a user-defined control class for the insertion of missing values and the corresponding method for `setNA()`.

are used to handle data generation, sampling, contamination and the insertion of missing values. Due to the use of control objects, switching from one simulation design to another requires only minimal programming effort. Developers can easily extend the existing framework with user-defined classes and methods. Guidelines for simulation studies in research projects can therefore be established by selecting or implementing control classes and agreeing upon parameter values, thus ensuring comparable results from different researchers. Based on the structure of the simulation results, an appropriate plot method is selected automatically. Hence **simFrame** is widely applicable for gaining insight into the quality of statistical methods. Furthermore, since the workload in statistical simulation is embarrassingly parallel, **simFrame** supports parallel computing using the package **parallel** to increase computational performance.

Future plans include to further develop the model-based data generation facilities and implement mixed simulation designs, to improve the support for small area estimation, as well as to extend the framework with different sampling methods and more specialized contamination and missing data models. Concerning large data sets, the incorporation of the package **ff** for memory-efficient storage may be investigated.

Acknowledgments

This work was partly funded by the European Union (represented by the European Commission) within the 7$^{\text{th}}$ framework programme for research (Theme 8, Socio-Economic Sciences and Humanities, Project AMELI (Advanced Methodology for European Laeken Indicators), Grant Agreement No. 217322). Visit http://ameli.surveystatistics.net for more information on the project.

Furthermore, we would like to thank two anonymous referees for their constructive remarks that helped to improve the package and the paper.

References

Adler D, Gläser C, Nenadic O, Oehlschlägel J, Zucchini W (2010). ***ff**: Memory-Efficient Storage of Large Atomic Vectors and Arrays on Disk and Fast Access Functions*. R package version 2.2-1, URL http://ff.r-forge.r-project.org.

Aitchison J (1986). *The Statistical Analysis of Compositional Data*. Chapman & Hall, London. ISBN 0-412-28060-4.

Aitchison J (1992). "On Criteria for Measures of Compositional Difference." *Mathematical Geology*, **24**(4), 365–379.

Aitchison J, Barceló-Vidal C, Martín-Fernández J, Pawlowsky-Glahn V (2000). "Logratio Analysis and Compositional Distance." *Mathematical Geology*, **32**(3), 271–275.

Alfons A (2013). ***simFrame**: Simulation Framework*. R package version 0.5.1, URL http://CRAN.R-project.org/package=simFrame.

Alfons A, Holzer J, Templ M (2010a). ***laeken**: Laeken Indicators for Measuring Social Cohesion*. R package version 0.1.3, URL http://CRAN.R-project.org/package=laeken.

Alfons A, Kraft S (2010). ***simPopulation**: Simulation of Synthetic Populations for Surveys based on Sample Data*. R package version 0.2, URL http://CRAN.R-project.org/package=simPopulation.

Alfons A, Kraft S, Templ M, Filzmoser P (2010b). "Simulation of Synthetic Population Data for Household Surveys with Application to EU-SILC." *Research Report CS-2010-1*, Department of Statistics and Probability Theory, Vienna University of Technology. URL http://www.statistik.tuwien.ac.at/forschung/CS/CS-2010-1complete.pdf.

Alfons A, Templ M, Filzmoser P (2010c). "Contamination Models in the R Package **simFrame** for Statistical Simulation." In S Aivazian, P Filzmoser, Y Kharin (eds.), *Computer Data Analysis and Modeling: Complex Stochastic Data and Systems*, volume 2, pp. 178–181. Minsk. ISBN 978-985-476-848-9.

Alfons A, Templ M, Filzmoser P (2010d). "An Object-Oriented Framework for Statistical Simulation: The R Package **simFrame**." *Journal of Statistical Software*, **37**(3), 1–36. URL http://www.jstatsoft.org/v37/i03/.

Alfons A, Templ M, Filzmoser P, Holzer J (2010e). "A Comparison of Robust Methods for Pareto Tail Modeling in the Case of Laeken Indicators." In C Borgelt, G González-Rodríguez, W Trutschnig, M Lubiano, M Gil, P Grzegorzewski, O Hryniewicz (eds.), *Combining Soft Computing and Statistical Methods in Data Analysis*, volume 77 of *Advances in Intelligent and Soft Computing*, pp. 17–24. Springer-Verlag, Heidelberg. ISBN 978-3-642-14745-6.

Alfons A, Templ M, Filzmoser P, Kraft S, Hulliger B (2009). "Intermediate Report on the Data Generation Mechanism and on the Design of the Simulation Study." *AMELI Deliverable 6.1*, Vienna University of Technology. URL http://ameli.surveystatistics.net.

Arnold K, Gosling J, Holmes D (2005). *The Java Programming Language*. 4th edition. Prentice Hall, Upper Saddle River. ISBN 978-0321349804.

Atkinson T, Cantillon B, Marlier E, Nolan B (2002). *Social Indicators: The EU and Social Inclusion*. Oxford University Press, New York. ISBN 0-19-925349-8.

Béguin C, Hulliger B (2008). "The BACON-EEM Algorithm for Multivariate Outlier Detection in Incomplete Survey Data." *Survey Methodology*, **34**(1), 91–103.

Burton A, Altman D, Royston P, Holder R (2006). "The Design of Simulation Studies in Medical Statistics." *Statistics in Medicine*, **25**(24), 4279–4292.

Chambers J (1998). *Programming with Data*. Springer-Verlag, New York. ISBN 0-387-98503-4.

Chambers J (2008). *Software for Data Analysis: Programming with R*. Springer-Verlag, New York. ISBN 978-0-387-75935-7.

Chambers J, Hastie T (1992). *Statistical Models in S*. Chapman & Hall, London. ISBN 9780412830402.

Cochran W (1977). *Sampling Techniques*. 3rd edition. John Wiley & Sons, New York. ISBN 0-471-16240-X.

Deville JC, Särndal CE, Sautory O (1993). "Generalized Raking Procedures in Survey Sampling." *Journal of the American Statistical Association*, **88**(423), 1013–1020.

Egozcue J, Pawlowsky-Glahn V, Mateu-Figueras G, Barceló-Vidal C (2003). "Isometric Logratio Transformations for Compositional Data Analysis." *Mathematical Geology*, **35**(3), 279–300.

EU-SILC (2004). "Common Cross-Sectional EU Indicators based on EU-SILC; the Gender Pay Gap." *EU-SILC 131-rev/04*, Working group on Statistics on Income and Living Conditions (EU-SILC), Eurostat, Luxembourg.

Fowler M (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd edition. Addison-Wesley. ISBN 978-0-321-19368-1.

Genz A, Bretz F (2009). *Computation of Multivariate Normal and t Probabilities*, volume 195 of *Lecture Notes in Statistics*. Springer-Verlag, New York. ISBN 978-3-642-01688-2.

Genz A, Bretz F, Miwa T, Mi X, Leisch F, Scheipl F, Hothorn T (2010). ***mvtnorm**: Multivariate Normal and t Distributions*. R package version 0.9-92, URL http://CRAN.R-project.org/package=mvtnorm.

Hill B (1975). "A Simple General Approach to Inference about the Tail of a Distribution." *The Annals of Statistics*, **3**(5), 1163–1174.

Hron K, Templ M, Filzmoser P (2010). "Imputation of Missing Values for Compositional Data using Classical and Robust Methods." *Computational Statistics & Data Analysis*, **54**(12), 3095–3107.

Hulliger B, Schoch T (2009a). "Robust Multivariate Imputation with Survey Data." 57th Session of the International Statistical Institute, Durban.

Hulliger B, Schoch T (2009b). "Robustification of the Quintile Share Ratio." New Techniques and Technologies for Statistics, Brussels.

Johnson M (1987). *Multivariate Statistical Simulation*. John Wiley & Sons, New York. ISBN 0-471-82290-6.

Kleiber C, Kotz S (2003). *Statistical Size Distributions in Economics and Actuarial Sciences*. John Wiley & Sons, Hoboken. ISBN 0-471-15064-9.

L'Ecuyer P, Leydold J (2005). "**rstream**: Streams of Random Numbers for Stochastic Simulation." *R News*, **5**(2), 16–20. URL http://CRAN.R-project.org/doc/Rnews/.

L'Ecuyer P, Simard R, Chen E, Kelton W (2002). "An Object-Oriented Random-Number Package with Many Long Streams and Substreams." *Operations Research*, **50**(6), 1073–1075.

Leisch F (2003). "**Sweave**, Part II: Package Vignettes." *R News*, **3**(2), 21–24.

Leydold J (2010). ***rstream**: Streams of Random Numbers*. R package version 1.2.5, URL http://statistik.wu-wien.ac.at/arvag/.

Little R, Rubin D (2002). *Statistical Analysis with Missing Data*. 2nd edition. John Wiley & Sons, New York. ISBN 0-471-18386-5.

Maronna R, Martin D, Yohai V (2006). *Robust Statistics*. John Wiley & Sons, Chichester. ISBN 978-0-470-01092-1.

Mateu-Figueras G, Pawlowsky-Glahn V, Egozcue J (2008). "The Normal Distribution in Some Constrained Sample Spaces." URL http://arxiv.org/abs/0802.2643.

Morgan B (1984). *Elements of Simulation*. Chapman & Hall, London. ISBN 0-412-24590-6.

Münnich R, Schürle J (2003). "On the Simulation of Complex Universes in the Case of Applying the German Microcensus." *DACSEIS research paper series No. 4*, University of Tübingen. URL http://w210.ub.uni-tuebingen.de/volltexte/2003/979/.

Münnich R, Schürle J, Bihler W, Boonstra HJ, Knotterus P, Nieuwenbroek N, Haslinger A, Laaksonen S, Eckmair D, Quatember A, Wagner H, Renfer JP, Oetliker U, Wiegert R (2003). "Monte Carlo Simulation Study of European Surveys." *DACSEIS Deliverables D3.1 and D3.2*, University of Tübingen. URL http://www.dacseis.de.

R Development Core Team (2010). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

Raghunathan T, Reiter J, Rubin D (2003). "Multiple Imputation for Statistical Disclosure Limitation." *Journal of Official Statistics*, **19**(1).

Rao J (2003). *Small Area Estimation*. John Wiley & Sons, Hoboken. ISBN 978-0-471-41374-5.

Ripley B (1987). *Stochastic Simulation*. John Wiley & Sons, New York. ISBN 0-471-81884-4.

Rossini A, Tierney L, Li N (2007). "Simple Parallel Statistical Computing in R." *Journal of Computational and Graphical Statistics*, **16**(2), 399–420.

Sarkar D (2008). *Lattice: Multivariate Data Visualization with R*. Springer-Verlag, New York. ISBN 978-0-387-75968-5.

Sarkar D (2010). ***lattice**: Lattice Graphics*. R package version 0.19-13, URL http://CRAN.R-project.org/package=lattice.

Schmidberger M, Morgan M, Eddelbuettel D, Yu H, Tierney L, Mansmann U (2009). "State of the Art in Parallel Computing with R." *Journal of Statistical Software*, **31**(1), 1–27. URL http://www.jstatsoft.org/v31/i01.

Sevcikova H, Rossini T (2009). ***rlecuyer**: R Interface to RNG with Multiple Streams*. R package version 0.3-1, URL http://CRAN.R-project.org/package=rlecuyer.

Templ M, Filzmoser P, Hron K (2009). "Robust Imputation of Missing Values in Compositional Data using the R-Package **robCompositions**." New Techniques and Technologies for Statistics, Brussels.

Templ M, Hron K, Filzmoser P (2010). ***robCompositions**: Robust Estimation for Compositional Data.* R package version 1.4.3, URL http://CRAN.R-project.org/package=robCompositions.

Tierney L, Rossini A, Li N (2009). "**snow**: A Parallel Computing Framework for the R System." *International Journal of Parallel Programming*, **37**(1), 78–90.

Tierney L, Rossini A, Li N, Sevcikova H (2008). ***snow**: Simple Network of Workstations.* R package version 0.3-3, URL http://CRAN.R-project.org/package=snow.

Tillé Y, Matei A (2009). ***sampling**: Survey Sampling.* R package version 2.3, URL http://CRAN.R-project.org/package=sampling.

Todorov V (2010). ***rrcov**: Scalable Robust Estimators with High Breakdown Point.* R package version 1.1-00, URL http://CRAN.R-project.org/package=rrcov.

Todorov V, Filzmoser P (2009). "An Object-Oriented Framework for Robust Multivariate Analysis." *Journal of Statistical Software*, **32**(3), 1–47. URL http://www.jstatsoft.org/v32/i03.

Troyanskaya O, Cantor M, Sherlock G, Brown P, Hastie T, Tibshirani R, Botstein D, Altman R (2001). "Missing Value Estimation Methods for DNA Microarrays." *Bioinformatics*, **17**(6), 520–525.

Urbanek S (2009). ***multicore**: Parallel Processing of R Code on Machines with Multiple Cores or CPUs.* R package version 0.1-3, URL http://www.rforge.net/multicore/.

Vandewalle B, Beirlant J, Christmann A, Hubert M (2007). "A Robust Estimator for the Tail Index of Pareto-Type Distributions." *Computational Statistics & Data Analysis*, **51**(12), 6252–6268.

Affiliation:

Andreas Alfons
Erasmus School of Economics
Erasmus Universiteit Rotterdam
PO Box 1738
3000DR Rotterdam, Netherlands
E-mail: alfons@ese.eur.nl
URL: http://people.few.eur.nl/alfons/