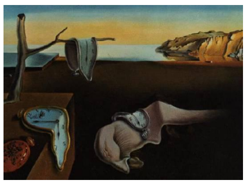

Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

Accelerating weighted random sampling without replacement

Kirill Müller
IVT, ETH Zurich

Abstract

Random sampling from discrete populations is one of the basic primitives in statistical computing. This article briefly introduces weighted and unweighted sampling with and without replacement. The case of weighted sampling without replacement appears to be most difficult to implement efficiently, which might be one reason why the R implementation performs slowly for large input. This paper presents four alternative implementations for the case of weighted sampling without replacement, with an analysis of their run time and correctness.

Keywords: Weighted sampling, performance, meta-analysis, R.

1. Introduction

Random sampling from discrete populations is one of the basic primitives in statistical computing. This paper focuses on a specific variant: sampling without replacement from a finite population with non-uniform weight distribution. One application for weighted sampling without replacement is the "Truncate-Replicate-Sample" method for stochastic conversion of positive real-valued weights to integer weights in the domain of spatial microsimulation (Lovelace and Ballas 2013). Further applications include market surveys, quality control in manufacturing, and on-line advertising (Efraimidis 2010).

Throughout this paper, the term *weight* refers to the relative probability that an item is sampled. A related problem, sampling from a population with given inclusion probabilities (without specifying an order) is beyond the scope of this paper.

First, different techniques for sampling from discrete populations are reviewed. Several implementations for sampling without replacement are discussed, this includes evaluation of run time performance and correctness. The paper concludes with suggestions for incorporating the findings into base R.

2. Sampling from discrete populations

Algorithm 1 is offered as a definition of sampling from discrete populations with or without replacement from arbitrary (including uniform) weight distributions.

Algorithm 1 sample($n, s, \text{replace}, p_i$)

```
Require: $n$: Size of the population
Require: $s$: Number of items to sample
Require: replace: true to request sampling with replacement
Require: $p_i$: Weight of each item for $i \in \{1, \ldots, n\}$
Ensure: Returns a vector $k_j \in \{1, \ldots, n\}$ for $j \in \{1, \ldots, s\}$ that contains the indexes of the
    items sampled
1: if $s = 0$ then
2:     return vector of length 0
3: end if
4: Randomly select $k$ so that, for all $i$, $\text{P}(k = i) = \frac{p_i}{\sum_j p_j}$
5: if not replace then
6:     $n \leftarrow n - 1$
7:     remove item $k$ from $p_i$
8: end if
9: return $k \oplus \text{sample}(n, s - 1, \text{replace}, p_i)$
```

From this definition, the following can be observed:

- Sampling with replacement appears to be a simpler problem than sampling without replacement, as the lines 5 and 8 in Algorithm 1 are not required.
- If all weights p_i are equal, the problem is simpler as well: The selection probability $\text{P}(i = k)$ of the sampled items in line 4 always equals $\frac{1}{n}$ and does not have to be computed explicitly.

In the framework of Algorithm 1, sampling without replacement with non-uniform weights seems to be the hardest problem. This intuition carries over to the more specialized algorithms for sampling with and without replacement, and with uniform or arbitrary weights, which are presented in the remainder of this section.

2.1. Sampling with replacement

The *with replacement* case corresponds to repeated selection of k from a fixed discrete weight distribution. The uniform case can be implemented easily by transforming the output of a random number generator that returns uniformly distributed floating-point numbers in $[0, 1)$. (Implementing such a random number generator is nontrivial in itself but outside the scope of this paper.)

More work is needed in the non-uniform case: Here, Walker's alias method (Walker 1977), which is also used in R, is an option. Assuming w.l.o.g. $\sum_j p_j = n$, it is possible to construct a subdivision (l_i, r_i, s_i) with $i, l_i, r_i \in \{1, \ldots, n\}$ and $0 < s_i \leq 1$ so that

$$p_i = \sum_{j:l_j=i} s_j + \sum_{j:r_j=i} (1 - s_j).$$

Sampling an item requires sampling from $\{1, \ldots, n\}$ (to choose i) and then sampling from $[0, 1)$ (to choose l_i or r_i): If the random number is less than s_i, item l_i is chosen, otherwise item r_i. (Figuratively, the probability mass given by p_i is distributed over n "boxes" so that the space in each box i is assigned to at most two items l_i and r_i. The share occupied by item l_i in box i is given by s_i. Some items may be distributed over several boxes. Sampling an item means selecting a box and choosing between the two items in this box.)

Walker's alias method is optimal, requiring only $O(n)$ preprocessing time (in a modification suggested by Vose (1991)). Hence, for non-uniform weights, the run time is at least $O(n+s)$, and the input size n will dominate unless $s \gg n$. More recently, Marsaglia, Tsang, and Wang (2004) have suggested a table-based method that seems to perform much faster in practice but expresses the weights as rationals with a fixed base and is therefore not usable directly for distributions with a large range. Shmerling (2013) presents a comprehesive review and suggests a general method suitable even for quasi-infinite ranges.

2.2. Sampling without replacement

In the *without replacement* case, each selected item is removed from the collection of candidate items. Again, the uniform case is much simpler. An array of size n, initialized with the natural sequence, can be used for storing the candidate items. The selection of the item corresponds to choosing an index at random in this array. Removal of an item with known index can be done in $O(1)$ time by simply replacing it with the last item in the array and truncating the array by one.

For the non-uniform case, lines 4 and 7 in Algorithm 1 can be implemented with a data structure that maintains a subdivision of an interval into n subintervals and allows lookups and updates. Walker's alias method seems to be ill-suited for this purpose, as each item potentially spreads over several "boxes", and an efficient update algorithm seems elusive. Wong and Easton (1980) propose a data structure similar to a heap that can be initialized in $O(n)$ time and supports simultaneous lookup and update in $O(\log n)$ time, the reader is referred to the original paper for details.

2.3. Sampling according to selection probabilities

Tillé (2006) defines a more rigorous framework for sampling algorithms from the perspective of the likelihood that a sample is selected based on a given sampling design. In the context of that framework, Algorithm 1 belongs to the class of "draw by draw" algorithms. For the application of sampling theory, the order of the selected elements is not important and usually ignored; in contrast, Algorithm 1 returns an ordered sequence of sampled elements.

3. Implementation

R offers reasonably efficient implementations for all cases except non-uniform sampling without replacement. The stock implementation for weighted random sampling without replacement requires $O(n \cdot s)$ run time, which is equivalent to $O(n^2)$ if $s = O(n)$. This paper explores alternative approaches: rejection sampling, one-pass sampling and reservoir sampling. Only the first can be described formally within the framework of Algorithm 1, however an actual implementation would use sampling *with* replacement as a subroutine. The last two are based on an arithmetic transformation of a weight distribution.

3.1. Rejection sampling

In the framework of Algorithm 1, rejection sampling corresponds to flagging sampled items as "invalid" (instead of removing them) in line 7, and repeating the sampling in line 4 until hitting a valid item. Note that the distribution of the result is not modified if invalid items are purged occasionally. This corresponds to the class of "rejective algorithms" in the framework of Tillé (2006).

Therefore, sampling without replacement can be emulated by repeated sampling with replacement, as shown in Algorithm 2. The general idea is to sample slightly more items than necessary (with replacement), and then to throw away the duplicate items. If the resulting sequence of items is shorter than requested, the result for a much smaller problem is appended. In Algorithm 2, duplicate items in the result of a sampling with replacement (line 1) correspond to invalid items in the rejection sampling, and the recursive call in line 7 corresponds to purging the invalid items.

```
**Algorithm 2** sample.rej($n, s, p_i$)
**Require:** $n$: Size of the population
**Require:** $s$: Number of items to sample
**Require:** $p_i$: Weight of each item for $i \in \{1, \dots, n\}$
**Ensure:** Returns a vector $k_j \in \{1, \dots, n\}$ for $j \in \{1, \dots, s\}$ that contains the indexes of the
    items sampled
 1: $k_i \leftarrow$ unique(sample($n$, expected.items($n, s$), **true**, $p_i$))
 2: $l \leftarrow$ length($k_i$)
 3: **if** $l \geq s$ **then**
 4:     **return** the first $s$ items of $k_i$
 5: **end if**
 6: remove items $k_i$ from $p_i$
 7: **return** $k_i \oplus$ sample.rej($n - l, s - l, p_i$)
```

Here, expected.items(n, s) is an estimate for the number of items that need to be drawn with replacement, so that the result can be expected to contain at least s unique items. (An incorrect estimate only affects the run time, not the correctness of the algorithm.) Note that, with expected.items$(n, s) = 1$ everywhere, Algorithms 1 and 2 are in fact identical. For a uniform distribution, it can be shown that the result has approximately s unique items in expectation with expected.items$(n, s) = n(H_n - H_{n-s}) = n \sum_{i=n-s+1}^{n} \frac{1}{i}$. This is an underestimate for non-uniform distributions. Nevertheless, the implementation in this package uses this estimate, capped at $2n$. As shown in Section 5, this algorithm performs worse than the alternatives shown in the following section in the majority of cases, but still outperforms the stock implementation for large values of n and s. Therefore, no tuning of the estimation of the number of expected items has been performed.

3.2. One-pass sampling

A particularly interesting algorithm has been devised only recently by Efraimidis and Spirakis (2006). In the simplest version (here referred to as *one-pass sampling*), it is sufficient to draw n random numbers, combine them arithmetically with the weight distribution p_i, and perform a partial sort to find the indexes of the s smallest items. Algorithm 3 is a modified version of Algorithm A in the original paper that operates on the logarithmic scale and uses multiplication instead of exponentiation for increased numeric stability.

Algorithm 3 sample.rank(n, s, p_i)

```
Require: $n$: Size of the population
Require: $s$: Number of items to sample
Require: $p_i$: Weight of each item for $i \in \{1, \ldots, n\}$
Ensure: Returns a vector $k_j \in \{1, \ldots, n\}$ for $j \in \{1, \ldots, s\}$ that contains the indexes of the
    items sampled
1: $r_i \leftarrow \mathrm{Exp}(1)/p_i$ for all $i \in \{1, \ldots, n\}$
2: return the positions of the $s$ smallest elements in $r_i$
```

The arithmetic transformation of the weight distribution is carried out in line 1. A sequence of i.i.d. samples from the exponential distribution with rate 1 is divided by the weights, the order of the results defines the sampling order. Intuitively, an item with a large weight has a larger probability of appearing earlier in this sorting order. Efraimidis and Spirakis (2006) prove that Algorithms 1 and 3 are equivalent.

The algorithm amazes with its elegance and simplicity. This also allows for almost trivial parallelization, provided that independent random number generators are available to each thread. Computational complexity is dominated by the partial sort (which can be implemented in $O(n + s \log n)$, or even in $O(n)$ for floating-point numbers (Terdiman 2000). However, the cost of generating n random variates may outweigh the cost for sorting even for moderately large values of s. The next subsection describes an extension to overcome this issue.

3.3. Reservoir sampling

Reservoir sampling with exponential jumps is a modified version of one-pass sampling. A reservoir of "active" items is maintained. Each generated random number decides how many input items are skipped until the current "least likely" item is removed from the reservoir. Algorithm 4 shows a verbal description, further details and formal proofs of correctness are beyond the scope of this paper and can be found in (Efraimidis and Spirakis 2006). Only $O(s \log \frac{n}{s})$ random numbers (in expectation) are needed with this extension, whereas the simple version always requires n random numbers. The exponential jumps method requires fewer updates of the reservoir (and therefore fewer random numbers and less run time) if the weights are arranged in descending order. In addition to drawing random numbers, the extraction of the smallest item from a priority queue (line 4) is the most expensive operation.

4. Implementation

The **wrswoR** package contains implementations for the two algorithms presented in the previous section, one R implementation of rejection sampling (Algorithm 2, denoted by *rej*), two implementations (R and C++) of one-pass sampling (Algorithm 3, *rank* and *crank*) and one C++ implementation of reservoir sampling with exponential jumps (*expj*, Algorithm 4). In

```
Algorithm 4 sample.expj($n, s, p_i$)
Require: $n$: Size of the population
Require: $s$: Number of items to sample
Require: $p_i$: Weight of each item for $i \in \{1, \ldots, n\}$
Ensure: Returns a vector $k_j \in \{1, \ldots, n\}$ for $j \in \{1, \ldots, s\}$ that contains the indexes of the
    items sampled
 1: Initialize reservoir with the first $s$ elements
 2: Set keys for these elements based on their weight and one random number per item
 3: while not all items processed do
 4:    Choose item with lowest key in the reservoir
 5:    Determine number of items to skip, based on this key and a random number
 6:    Replace item with lowest key with current item in the reservoir
 7:    Set the new item's key based on its weight and a random number
 8: end while
 9: return  Items in reservoir sorted by their key
```

the package, the corresponding functions are prefixed with `sample_int_`. The **Rcpp** package (Eddelbuettel and François 2011; Eddelbuettel 2013) is used to generate the glue between R and C++.

The R implementations are very similar to the pseudocode: As an example, the *rank* implementation is shown below.

```
function (n, size, prob)
{
    .check_args(n, size, prob)
    head(order(rexp(n)/prob), size)
}
```

The function arguments correspond to those of Algorithms 1 to 4: `size` is the s argument, and `prob` is the p_i argument.

The *crank* implementation has been somewhat optimized for cache efficiency. Due to its relative complexity, the *expj* implementation is kept very close to the pseudocode in the original paper, still this function also operates on the logarithmic scale for numeric stability. The transformation works in a fashion very similar to that of Algorithm 3.

All functions share the same interface. Compared to the base R function `sample.int()`, the `replace` parameter has been removed, and `prob` cannot be `NULL`. To simplify testing the new routines against the R implementation, a wrapper function `sample_int_R()` is provided:

```
function (n, size, prob)
{
    sample.int(n, size, replace = FALSE, prob)
}
```

The remainder of this paper presents performance characteristics and a validation of the new implementations.

5. Performance

This section presents run time tests for various combinations of input parameters, attempts to provide guidance when to choose which implementation, and discusses the correctness of the implementation. All test results shown in this section are based on data available in the **wrswoR.benchmark** package.

5.1. Input parameters

The run time tests used different values for the function arguments n, s and *prob*. Instead of directly specifying s, it is given as a proportion of n, denoted by $r = \frac{s}{n}$. The following weight distributions (used for p_i) were tested:

uniform $p_i = 1$ everywhere

linear Sequence from 1 to n ($\{p_i\} = \{1, \ldots n\}$), *ascending* ($\nearrow$), *descending* ($\searrow$) and *shuffled* ($\rightsquigarrow$)

geometric Starting at 1, the weight is multiplied with a constant α for each step ($p_{i+1} = \alpha p_i$, *ascending*, *descending*, and *shuffled*); the constant is chosen so that both minimal and maximal weights and the sum of weights is still representable as a floating-point number.

The geometric case is very extreme and unlikely to occur in practice, it is included here to test potential limitations of the implementations.

5.2. Run time

The run time was measured using the **microbenchmark** package (Mersmann 2015) in block order with a warmup of 10 iterations using the default 100 iterations. The tests ran on a single core of an Intel Xeon CPU X5680 clocked at 3.33 GHz with 12 MB cache, running Red Hat Enterprise Linux Server release 7.2, R version 3.2.3, and version 0.4 of the **wrswoR** package.

Figure 1 presents an overview of the median run time for different input sizes, output size ratios, weight distributions and implementations. The R implementation is outperformed by all other implementations for $n \approx 1000$, in many cases even for much smaller inputs. In the log-log scale used here, the slope of the curves translates to computational complexity; the steeper slope for the R implementation corresponds to its quadratic complexity compared to the only slightly superlinear complexity of the other algorithms. No data were obtained for the R implementation if the computation would have taken too long, this is reflected by a premature ending of the corresponding curves in Fig. 1.

As expected, the *expj* implementation is among the fastest, especially for $r \ll 1$. In the case $r = 0.01$ for the *geometric ascending* distribution, the new implementations win only by a margin; in particular, the run time of *expj* depends on the ordering of the weights which is unfavorable here.

The *rej* and *rank* implementations exhibit initial costs on the sub-millisecond scale even for small input sizes, probably due to the fact that both are implemented purely in R. In addition, the *rej* code is by far the slowest (but still faster than the stock implementation) for *geometric*

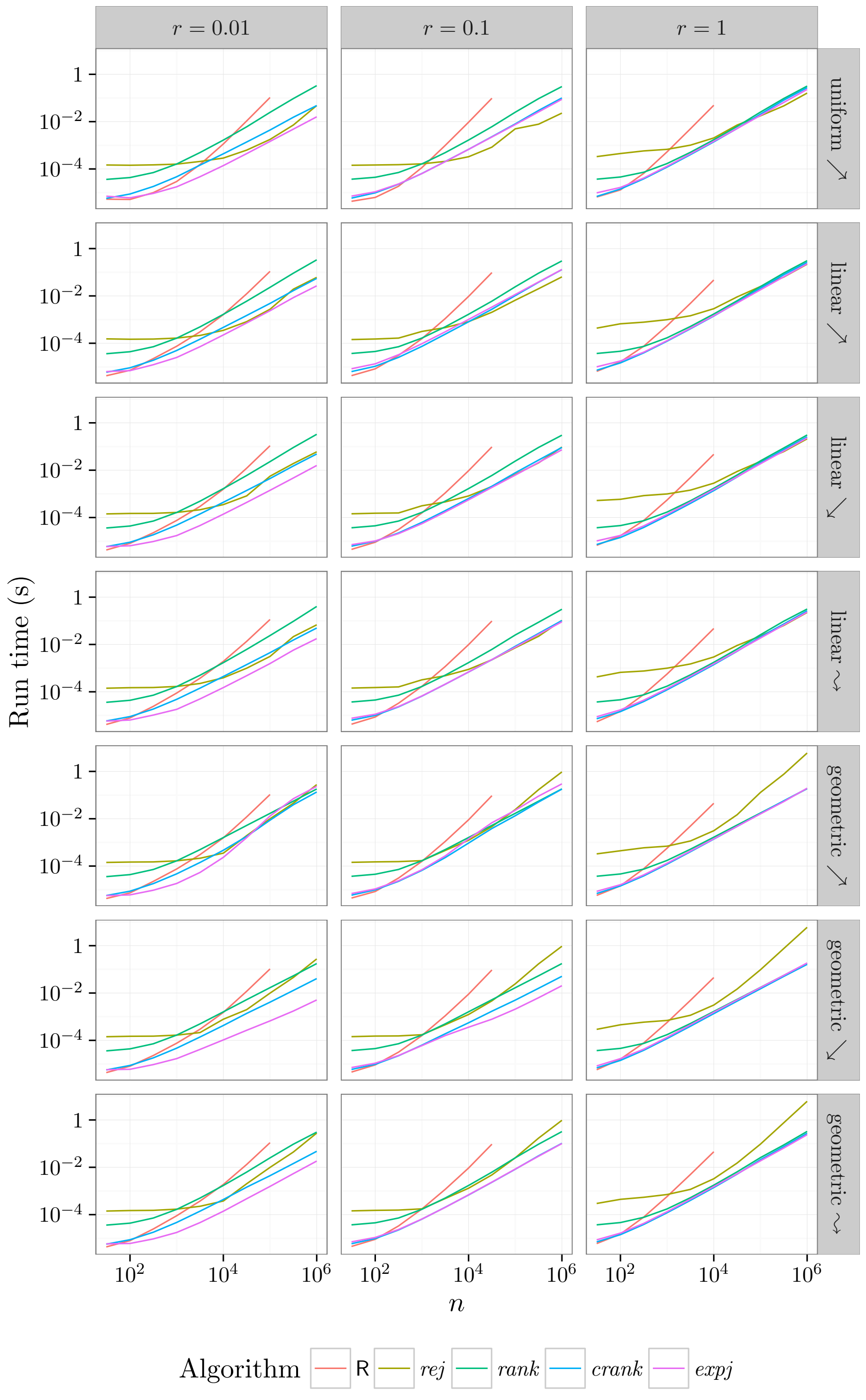

Figure 1: Median run times

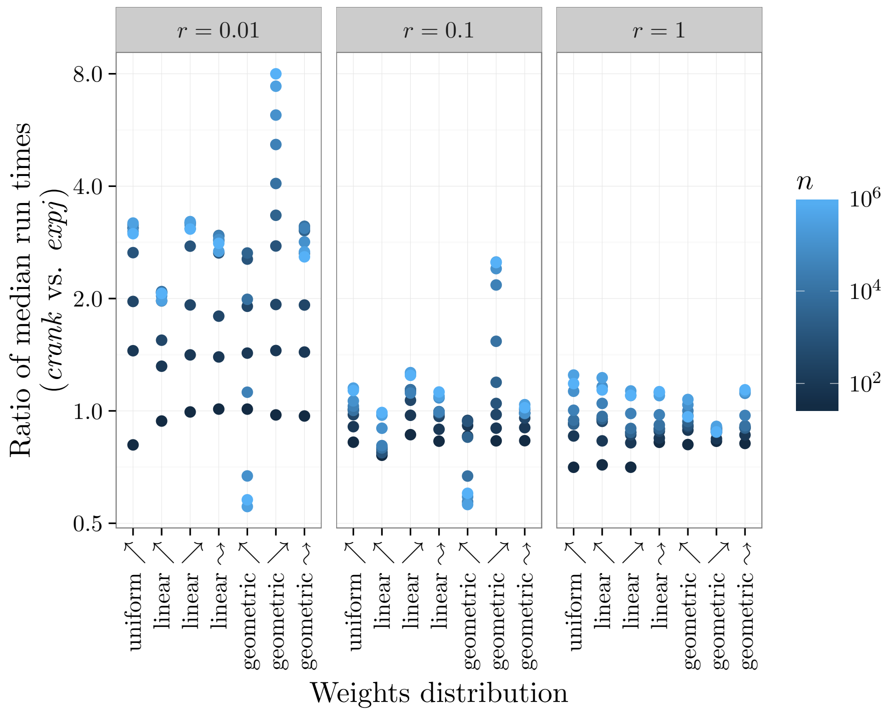

Figure 2: Comparison of *crank* and *expj* run times

distributions, because in each step only a tiny fraction of items have a non-negligible weight, and hence most sampled items are rejected as duplicates (line 1 of Algorithm 2).

Figure 2 compares run times for *crank* and *expj* for the different weight distributions, values above 1 mean that *expj* is faster. The *expj* implementation seems to perform better than *crank* if r is small or n is large. For the pathological *geometric* cases, the run time differences between *ascending* and *descending* weights are substantial for small r. The advantage of the *expj* code for $r = 1$ and large n is surprising and can only be explained with differences in run time between partial sort (which is used for *crank*) and priority queue (for *expj*).

For the break-even analysis, *expj* is compared to the stock implementation in Fig. 3 for linear ascending weights. The *expj* implementation can be up to about 2 times slower than the stock implementation, for absolute run times of around 10 microseconds for $n = 100$. It is remarkable that the relative performance of *expj* is worst with $r = 0.1$ in this case. The relative slowness of the stock implementation for the case $r = 0.01$ is due to a mandatory pre-sorting of weights using heap sort even for $s = 1$, which is not required for *expj*.

Figure 4 shows a more detailed break-even point analysis for a larger choice for r and for all weight distributions tested. Compared to *expj*, the stock implementation performs best with a uniform weight distribution, offsetting the break-even point to just below 500 for the best choice of $r \approx 0.1$. In other words, for $n < 500$ and $s = \lceil 0.1n \rceil$, the stock implementation is still the best choice in the case of a uniform or near-uniform distribution, with a speedup of at most 2.12.

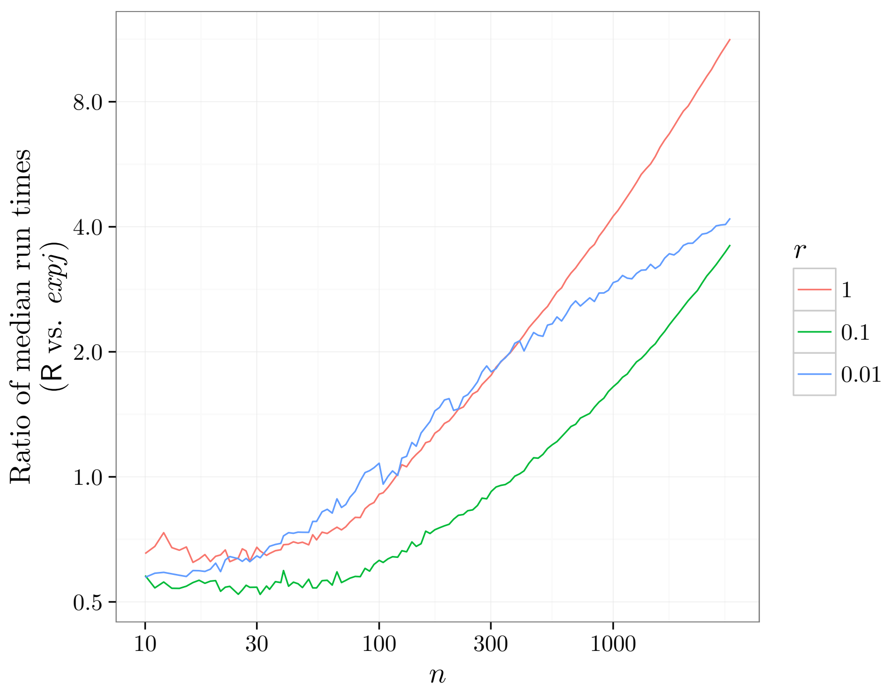

Figure 3: Comparison of R and *expj* run times for linear ascending weights

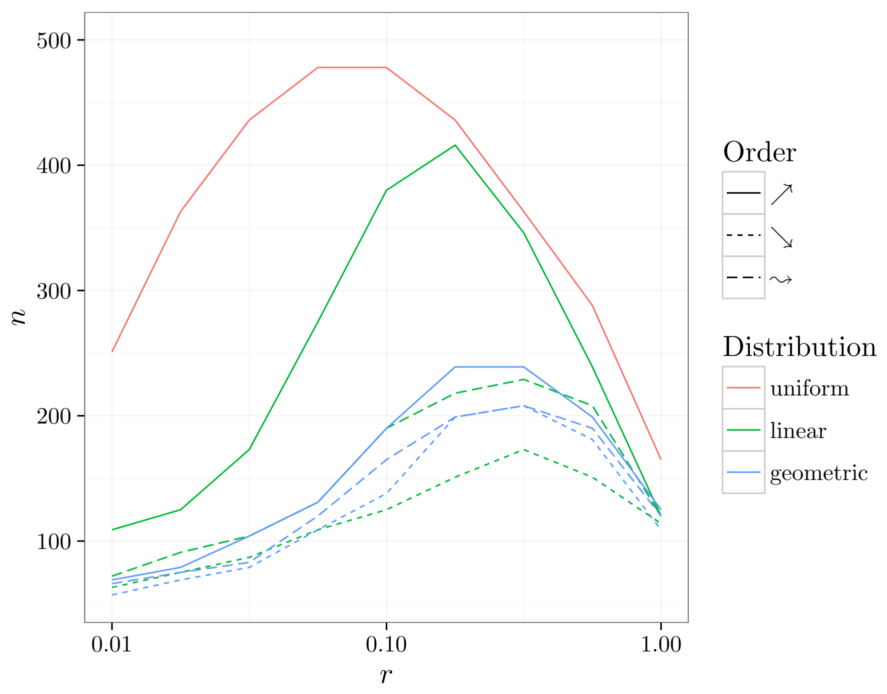

Figure 4: Break-even point of R and *expj* run times

6. Correctness

This section aims at validating the new implementations. A correct implementation should satisfy the following criteria:

1. All output items are between 1 and n.
2. Each item occurs at most once in the output.
3. For given parameters n, s and p_i, the probability that item i is at position j in the output (with $1 \leq i \leq n$ and $1 \leq j \leq s$) is identical for the implementation under test and the stock implementation.

Verifying these criteria seems to be challenging due to the stochasticity of the algorithms. The first two can be simply checked by observing the output. The following subsection describes a procedure for checking the third criterion.

6.1. Methodology

For fixed i and j and for fixed parameters n, s, and p_i, each call to the sampling routine is a Bernoulli trial with fixed success probability $\pi_{i,j}$. Repeated sampling leads to an i.i.d. sequence of Bernoulli trials. In general, computing the exact value of $\pi_{i,j}$ for large j seems to require considerable computational resources. Therefore, the value of $\pi_{i,j}$ is assumed unknown, and only the equality of the proportions is tested for the different implementations using a two-sided test for equal proportions (essentially a χ^2 test, implemented by the `prop.test()` function). The correctness check is performed as follows:

- The parameters n, s, and p_i, and the implementation under test, are fixed.
- For both the tested and the stock implementation, N random samples without replacement are drawn and recorded.
- For all i and j, the number of samples where item i is in position j (denoted by $f_{i,j}$) is computed.
- The counts are tested for equality of proportions, yielding a p-value for each tuple (i, j).

In this setting, for fixed (i, j), the p-value is itself a random variable that is distributed uniformly over $(0, 1]$ under the null hypothesis of equal proportions (i.e., if the tested implementation is correct). On the other hand, if the implementation is faulty, the rejection rate for the null hypothesis will be large, and a substantial share of the p-values will be very close to 0. While this procedure does not constitute a proof of correctness, it offers a means to automatically test the implementations for nontrivial errors. A similar procedure (using a visual representation with violin plots) caught an implementation error in the *expj* code that occurred only in the case $1 < s < n$.

To assert the sensitivity of the testing procedure, a faulty implementation was simulated by passing altered weights to R's implementation. The modification consists of updating

$$p'_i := p_i \cdot \left(1 + \text{skew} \cdot \frac{i-1}{n-1}\right),$$

where a skew of zero means no change, and a skew of 1% corresponds to relative differences increasing between 0 % and 1 %.

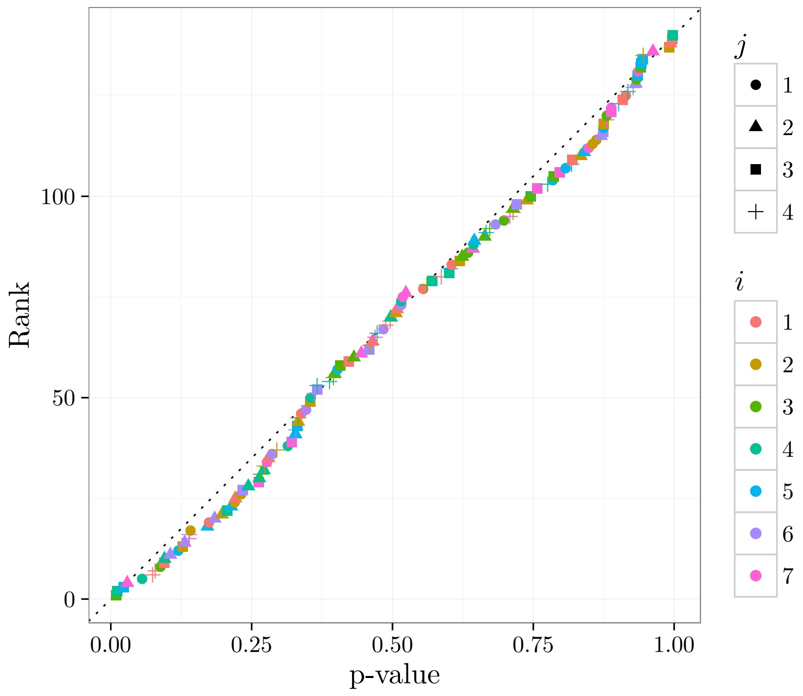

Figure 5: Schweder plot for p-values resulting from comparing the *crank* and R implementations

The test for equal proportions can be substituted by Fisher's exact test, which tends to produce lower p-values and therefore is usually more powerful than the test for equal proportions. However, Fisher's exact test has $O(N)$ complexity, because it evaluates the density of the hypergeometric distribution on a support of the order of N. Using this test would have been prohibitive in the setting described here.

6.2. Example

Figure 5 shows a Schweder plot (Schweder and Spjøtvoll 1982) of the p-values resulting from an experiment that draws $N = 2^{22}$ samples for $n = 7$, $s = 4$, and a geometric weight distribution with $\alpha = 1.08$, using all five implementations. Different values of i and j are denoted with different colors and shapes. The theoretical distribution is shown as a dotted line, and aligns very well with the observed p-values. Fisher's combined probability test is a meta-analysis method that combines multiple p-values (from different but related studies) into one; it is implemented in the **metap** package (Dewey 2014). For this particular run of the experiment, Fisher's method cannot reject the null hypothesis of uniformity ($p = 0.896$). As an example for a positive test, Fig. 6 shows results for the same experiment, now substituting the stock implementation with a faulty one with skew = 0.25 %. Despite the relative similarity of the weight distributions, the distribution of the p-values deviates substantially from the uniform distribution, with more p-values close to zero than expected. Here, Fisher's method detects significant, although not overwhelming, evidence against the null hypothesis ($p = 0.0183$).

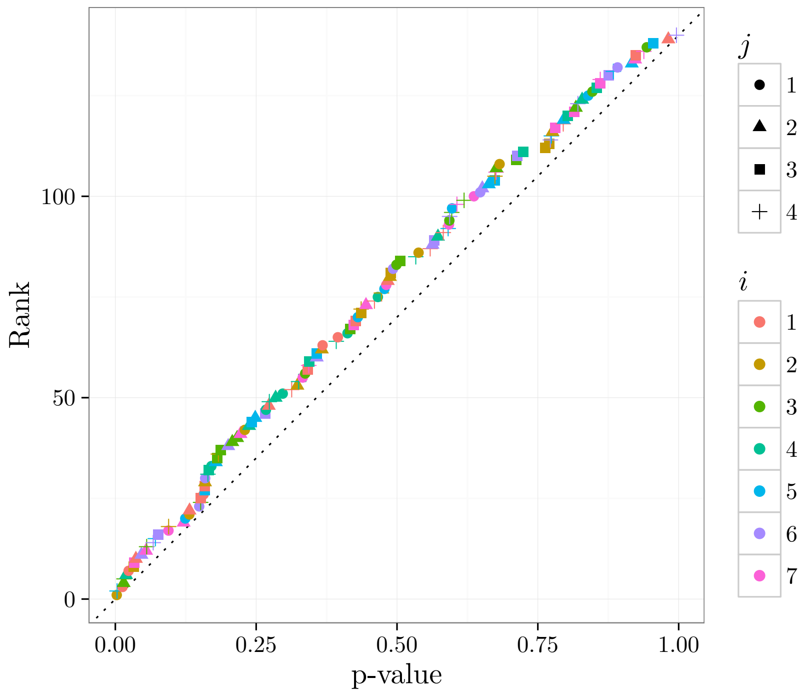

Figure 6: Schweder plot for p-values resulting from comparing the R implementation with a skewed version of itself

6.3. Results

A fairly comprehensive test also has been carried out, covering all $n \in \{2, \ldots 80\}$, a subset of $s \in \{1, \ldots, n\}$, and all (i, j). For each combination, the cell frequencies $f_{i,j}$ were collected for all new implementations, and for the stock implementation with and without altered weights (using skew values between 0.25 % and 16 %), for N ranging from 2^{10} to 2^{24} (only powers of 2). Each cell frequency was compared to that of the stock implementation. This resulted in around 5×10^8 p-values, which were again combined using Fisher's method.

Figure 7 shows the results of the meta-analysis separately for each N and for each (supposedly correct or faulty) implementation. Comparing the stock implementation to itself (using different random seeds) resulted in a p-value of almost 1 for all N, the same holds for all new codes. On the other hand, all skews tested led to strong rejection of the correctness hypothesis (p-value effectively 0) sooner or later; as expected, the smaller the skew, the larger the N that is required for rejection.

This comparison is less sensitive to implementation errors that occur only for specific arguments (e.g., if an implementation behaves as expected except if n is a power of 2). To catch such deficiencies, it is helpful to analyze finer aggregates of the p-values. Figure 8 shows combined p-vaules separately for all pairs of n and N when comparing each new code to the stock implementation. Some p-values are in the range of $(0.01, 0.1]$ or even $\left(10^{-4}, 0.01\right]$, but this can be expected due to the uniform distribution of the p-values under the null hypothesis. The plot in Fig. 9 is similar, but shows the p-values that result from comparing the stock implementation with a skewed version of itself, for different skews. Here, for all skews except 0, the combined p-value approaches zero sooner or later as N increases.

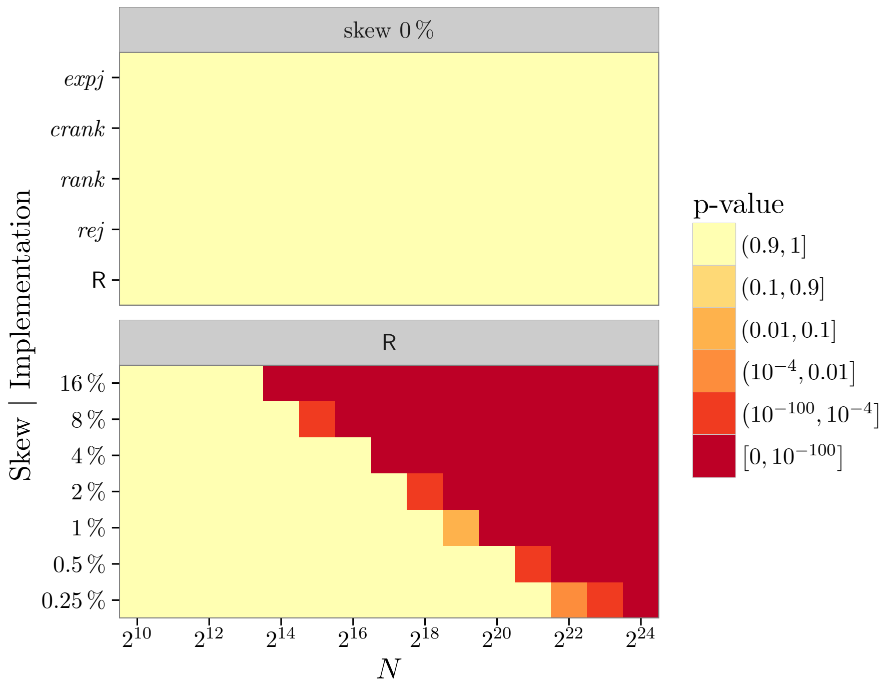

Figure 7: Combining p-values for a comprehensive test

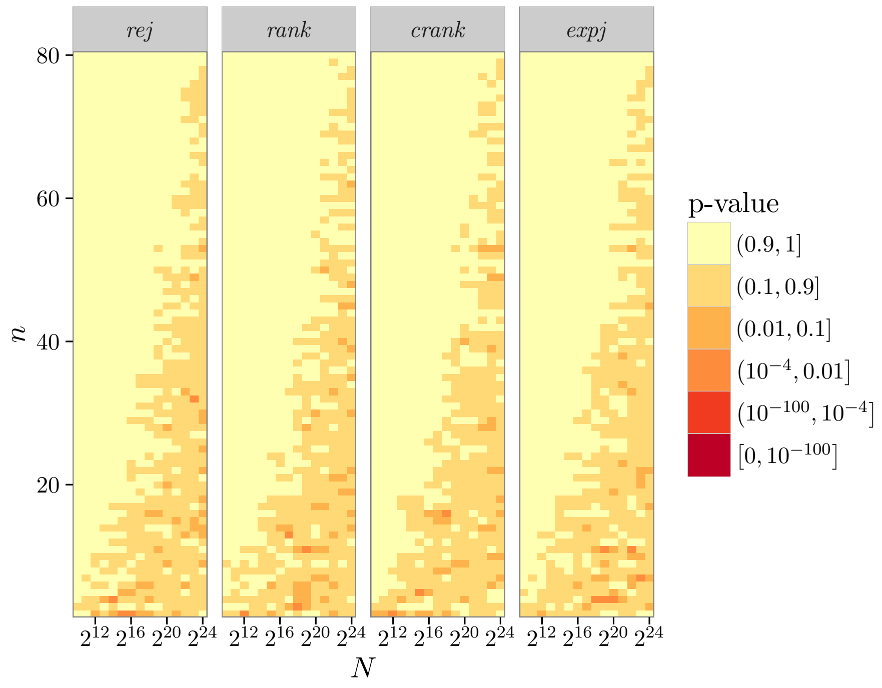

Figure 8: Combined p-values for different values of n and N, resulting from comparing each new code to the stock implementation

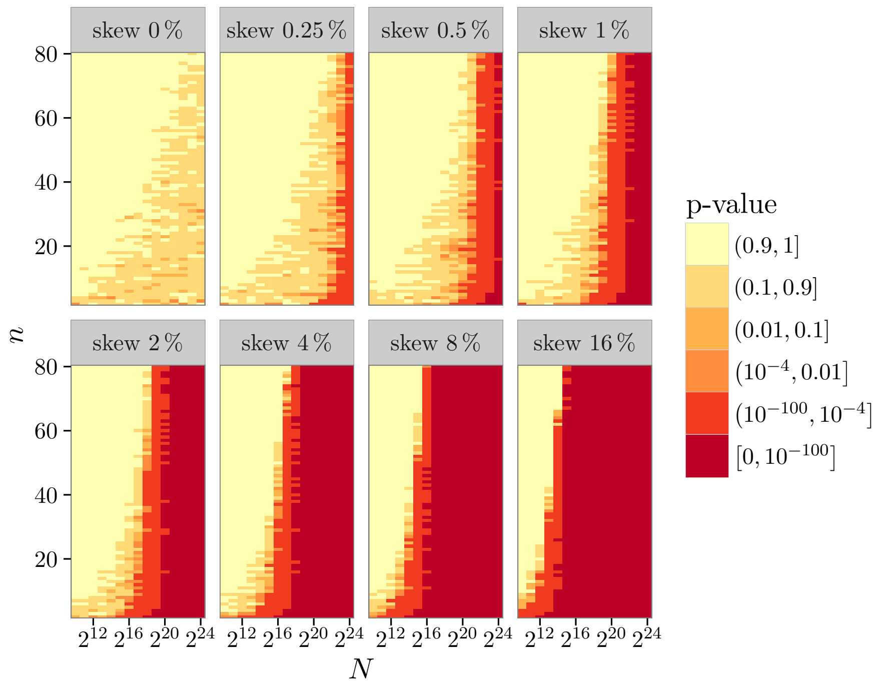

Figure 9: Combined p-values for different values of n and N, resulting from comparing the stock implementation to a skewed version of itself

7. Conclusions and future work

This paper describes four new implementations for weighted random sampling without replacement in R. The new implementations, even those written in pure R, clearly outperform the one provided by the **base** package if the number of items to choose from is just above 1000. Each of the algorithms presented here has its advantages:

- Rejection sampling is a simple and straightforward method that builds upon weighted sampling with replacement.
- One-pass sampling can be parallelized easily.
- Reservoir sampling with exponential jumps is fast even for degenerate weight distributions, and economical in its use of random numbers.

In particular, reservoir sampling with exponential jumps (Efraimidis and Spirakis 2006) requires just about double the time of the stock implementation in the worst case, code optimization (such as using a cache-efficient heap structure for the priority queue) might help further reduce this threshold or even remove it entirely. Reservoir sampling performs best if the weights are pre-sorted in descending order. An optional sorting step could be provided for convenience.

For validation, the new implementations have been compared with the stock implementation by counting the number of occurrences for each item and each possible position in a large number of runs, and testing the null hypothesis of equal proportions. This yields a massive amount of p-values, which can be combined using Fisher's method, a meta-analysis technique.

The validation methodology is able to clearly detect an emulated implementation error, which consisted of skewing the input frequency distribution in a predefined fashion, whereas no difference between the new and the stock implementations could be measured. So far, the detection of non-systematic errors or other failure modes has not been tested.

In order to include a faster sampling algorithm into base R, an implementation in C seems necessary. Other platforms for scientific computing, such as Python or Julia, would also benefit if this implementation was provided in an open-source library with a documented interface.

For the current implementation in R, a user might not expect a natural operation such as random sampling to take excessive time, without the ability to interrupt it. Calling `R_CheckUserInterrupt()` every 10^7 or so operations in the current implementation would at least save the unaware user the frustration of a lost workspace.

The algorithms presented here generate an ordered sample of items based on relative weights. If the relative importance is instead given as inclusion probabilities, and the order of the items is irrelevant, e.g., as in the application of survey sampling, the `UPxxx()` functions in the **sampling** package (Tillé and Matei 2015) offer a viable alternative.

8. Acknowledgments

This work would have been much more difficult without the **BatchJobs** and **BatchExperiment** packages (Bischl et al. 2015). The **dplyr** (Wickham and Francois 2015) and **tidyr** packages (Wickham 2016) helped processing the data; the plots were created with **ggplot2** (Wickham 2009) and rendered with **tikzDevice** (Sharpsteen and Bracken 2016). This document was created using **knitr** (Xie 2016; Xie 2015; Xie 2014) and **rmarkdown** (Allaire, Cheng, et al. 2016), based on a template provided by **rticles** (Allaire, R Foundation, et al. 2016).

References

Allaire, JJ, Joe Cheng, Yihui Xie, Jonathan McPherson, Winston Chang, Jeff Allen, Hadley Wickham, Aron Atkins, and Rob Hyndman. 2016. *Rmarkdown: Dynamic Documents for R*. https://CRAN.R-project.org/package=rmarkdown.

Allaire, JJ, R Foundation, Hadley Wickham, Journal of Statistical Software, Yihui Xie, Ramnath Vaidyanathan, Assocation for Computing Machinery, et al. 2016. *Rticles: Article Formats for R Markdown*. https://CRAN.R-project.org/package=rticles.

Bischl, Bernd, Michel Lang, Olaf Mersmann, Jörg Rahnenführer, and Claus Weihs. 2015. "BatchJobs and BatchExperiments: Abstraction Mechanisms for Using R in Batch Environments." *Journal of Statistical Software* 64 (11): 1–25. http://www.jstatsoft.org/v64/i11/.

Dewey, Michael. 2014. *Metap: Meta-Analysis of Significance Values*. https://CRAN.R-project.org/package=metap.

Eddelbuettel, Dirk. 2013. *Seamless R and C++ Integration with Rcpp*. New York: Springer.

Eddelbuettel, Dirk, and Romain François. 2011. "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software* 40 (8): 1–18. http://www.jstatsoft.org/v40/i08/.

Efraimidis, Pavlos S. 2010. "Weighted Random Sampling over Data Streams." *ArXiv:1012.0256*

[Cs], December.

Efraimidis, Pavlos S., and Paul G. Spirakis. 2006. "Weighted Random Sampling with a Reservoir." *Information Processing Letters* 97 (5): 181–85. doi:10.1016/j.ipl.2005.11.003.

Lovelace, Robin, and Dimitris Ballas. 2013. "'Truncate, Replicate, Sample': A Method for Creating Integer Weights for Spatial Microsimulation." *Computers, Environment and Urban Systems* 41 (September): 1–11. doi:10.1016/j.compenvurbsys.2013.03.004.

Marsaglia, George, Wai Wan Tsang, and Jingbo Wang. 2004. "Fast Generation of Discrete Random Variables." *Journal of Statistical Software* 11 (1): 1–11. doi:10.18637/jss.v011.i03.

Mersmann, Olaf. 2015. *Microbenchmark: Accurate Timing Functions*. `https://CRAN.R-project.org/package=microbenchmark`.

Schweder, T., and E. Spjøtvoll. 1982. "Plots of P-Values to Evaluate Many Tests Simultaneously." *Biometrika* 69 (3): 493–502. doi:10.1093/biomet/69.3.493.

Sharpsteen, Charlie, and Cameron Bracken. 2016. *TikzDevice: R Graphics Output in LaTeX Format*. `https://github.com/yihui/tikzDevice`.

Shmerling, Efraim. 2013. "A Range Reduction Method for Generating Discrete Random Variables." *Statistics & Probability Letters* 83 (4): 1094–9. doi:10.1016/j.spl.2013.01.002.

Terdiman, Pierre. 2000. "Radix Sort Revisited." *Online Paper*. `http://www.codercorner.com/RadixSortRevisited.htm`.

Tillé, Yves. 2006. "Sampling Algorithms." In *Sampling Algorithms*, 31–39. Springer Series in Statistics. Springer New York.

Tillé, Yves, and Alina Matei. 2015. *Sampling: Survey Sampling*. `https://CRAN.R-project.org/package=sampling`.

Vose, M.D. 1991. "A Linear Algorithm for Generating Random Numbers with a Given Distribution." *IEEE Transactions on Software Engineering* 17 (9): 972–75. doi:10.1109/32.92917.

Walker, Alastair J. 1977. "An Efficient Method for Generating Discrete Random Variables with General Distributions." *ACM Transactions on Mathematical Software (TOMS)* 3 (3): 253–56. `http://dl.acm.org/citation.cfm?id=355749`.

Wickham, Hadley. 2009. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. `http://had.co.nz/ggplot2/book`.

———. 2016. *Tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions*. `https://CRAN.R-project.org/package=tidyr`.

Wickham, Hadley, and Romain Francois. 2015. *Dplyr: A Grammar of Data Manipulation*. `https://CRAN.R-project.org/package=dplyr`.

Wong, C. K., and M. C. Easton. 1980. "An Efficient Method for Weighted Sampling Without Replacement." *SIAM Journal on Computing* 9 (1): 111–13. doi:10.1137/0209009.

Xie, Yihui. 2014. "Knitr: A Comprehensive Tool for Reproducible Research in R." In *Implementing Reproducible Computational Research*, edited by Victoria Stodden, Friedrich Leisch, and Roger D. Peng. Chapman; Hall/CRC. `http://www.crcpress.com/product/isbn/9781466561595`.

———. 2015. *Dynamic Documents with R and Knitr*. 2nd ed. Boca Raton, Florida: Chapman; Hall/CRC. `http://yihui.name/knitr/`.

———. 2016. *Knitr: A General-Purpose Package for Dynamic Report Generation in R*. `http://yihui.name/knitr/`.

Affiliation:

Kirill Müller
IVT, ETH Zurich
Stefano-Franscini-Platz 9, CH-8093 Zürich
E-mail: `kirill.mueller@ivt.baug.ethz.ch`
URL: `http://www.ivt.ethz.ch`

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume VV, Issue II *Submitted:* 2016-02-22
MMMMMM YYYY *Accepted:* yyyy-mm-dd