# Exact and Approximate Sampling from Categorical and Multinomial Distributions

Michael Colavita
Harvard College

Garrett Tanzer
Harvard College

**Abstract**

We present a broad survey of algorithms for exact and approximate sampling from categorical and multinomial distributions; we can view these problems as variations of the $\ell_1$ sampling problem, where we have a vector $a \in \mathbb{N}^k$ subject to streaming updates and wish to sample elements $x$ with $\Pr[x = i] \propto a_i$. We provide practical C++ implementations and experimental performance evaluations of these algorithms. Our main theoretical contribution is to improve the best beta-based multinomial sampling algorithm from $O(k \log n)$ to $O(k \log \log n)$ time, and our experimental contribution is to concretely implement and compare the state-of-the-art categorical sampling algorithms for the first time. Our code and testing apparatus are publicly available at sampling.

## 1 Introduction

Probability mass functions (pmfs) are a natural primitive in statistics. Given a table of probabilities corresponding to $k$ categories, the problem of drawing a single element is equivalent to sampling from the categorical distribution defined by that pmf, and the problem of counting the number of times each category is drawn in $n$ categorical samples is equivalent to sampling from its multinomial distribution. These fundamental statistical operations have myriad practical applications in computer science, ranging from more overtly relevant areas like machine learning, genetic algorithms, and Monte Carlo simulation to less apparently related systems problems like load-balancing, anomaly detection, and scheduling.

In the traditional setting, this probability table can be defined as a vector of nonnegative counts $a \in \mathbb{N}^k$, and an element $x$ is sampled from this vector such that $\Pr[x = i] = \frac{a_i}{\|a\|_1}$. We consider both the static case, where we sample from a fixed vector, and the dynamic case, where we sample from a vector as it is updated over time. We will describe and experimentally evaluate several algorithms that can efficiently and exactly draw an unlimited number of samples from the categorical or multinomial distributions defined by this vector, given access to $\Omega(k)$ working memory.

However, as data sets balloon in size, it becomes infeasible to use even linear working memory in the input size; in practice, it is common that we can only afford a single pass over data persisted across multiple hard disks, or real-time access to a stream of ephemeral data.

This motivates the use of sketching algorithms, which perform an approximate version of the same task using only $o(k)$ memory, or preferably $O(\log^c k)$ for some constant $c$, where $k$ is so large that the probability vector is never itself materialized. We will survey a selection of algorithms that can sample a single element $x$ with probability $\Pr[x = i] = (1 \pm \epsilon)\frac{a_i}{\|a_i\|} + O(n^{-c})$, with probability $\delta$ of catastrophic failure. In the streaming literature, this problem is a special case of the general $\ell_p$ sampling problem, in which samples are drawn with probability $\propto a_i^p$ for $p \geq 0$.

Our novel contributions are as follows. We provide the first known implementation of Matias et al.'s [10] state-of-the-art algorithm for dynamic categorical sampling, and evaluate it across several synthetic benchmarks. We also present an improvement to the class of algorithms pioneered by Relles [15] for multinomial sampling that uses draws from the beta distribution as a randomness source. At a high level, we improve Relles' $O(k \log n)$ runtime to $O(k \log \log n)$ by performing interpolation search rather than binary search at a key step in the algorithm and modifying the bookkeeping across searches to maintain correctness. This new algorithm substantially improves real-world performance of this class of algorithms, but falls short of the state-of-the-art using other approaches.

We will proceed in order through exact categorical sampling, exact multinomial sampling, approximate categorical sampling ($\ell_1$ sampling), and some discussion on why approximate multinomial sampling is not a studied problem, before finally concluding.

# 2 Categorical Sampling

## 2.1 Background

We will work in the strict turnstile setting, where a vector $a \in \mathbb{N}^k$ receives updates in the form of tuples $(i, c)$ representing the assignment $a_i \leftarrow a_i + c$, with the stipulation that at all points in time, all $a_i$ are nonnegative.

The problem of *categorical sampling*, or equivalently *exact $\ell_1$ sampling*, is to draw an unlimited number of independent and identically distributed samples $x$ according to the rule $\Pr[x = i] = \frac{a_i}{\|a\|_1}$. The categorical distribution is a generalization of the better-known Bernoulli distribution, in which $k = 2$.

By $O_{\mathbb{E}}(\dots)$, we denote *expected* $O(\dots)$, and by $O_A(\dots)$ we denote *amortized* $O(\dots)$.

## 2.2 Algorithms

We first describe the exact categorical sampling algorithms of interest at a conceptual level. For more complete detail, see the cited original papers or our fully specified implementations. All prominent algorithms use $\Theta(k)$ space.

### 2.2.1 The Alias Method

One class of categorical sampling algorithms is the alias method, pioneered by Walker [19]. This technique uses two tables of size $k$ called the *probability* table $F$ and the *alias* table

$A$, which are initialized by the particular algorithm. We sample by selecting $i$ uniformly at random in $[0, k-1]$, then outputting $i$ with probability $F_i$ and $A_i$ with probability $1-F_i$. The tables are constructed in such a way that the probability of outputting $i$ remains unchanged from the original pmf.

Vose [18] generates these tables in $O(k)$ time, an improvement over the naive $O(k \log k)$. At a high level, he does so by first separating the probability values into two buckets—*large* for values over $1/k$ and *small* for values under it—and then subdividing the large values to complement each small value in $F$ (and so indices with large probabilities will appear multiple times in $A$, while those with small probabilities will never appear).

This achieves sampling in $O(1)$ time with precomputation in $O(k)$ time, but there is no obvious way to perform updates in less than $O(k)$ time. The algorithm as described is also not numerically stable, but can be modified to work practically without harming asymptotic complexity [16].

### 2.2.2 Rejection Sampling Methods

The general idea of rejection sampling is that we can sample according to a density $f(x)$ by finding another density $g(x)$ such that $f(x) \leq cg(x)$ for some constant $c$, then sampling $u \sim \text{Unif}[0, 1)$ and $y \sim g(x)$ until $u < f(y)/cg(y)$. The expected number of iterations before reaching an acceptable value is $c$.

Rajasekaran and Ross [14] use this technique to sample from a limited class of streaming probability vectors in $O_{\mathbb{E}}(1)$ time, with $\Theta(1)$ updates as well. Specifically, they assume that there are *a priori* known lower and upper bounds on the entries of the probability vector. Using this condition, they essentially modify the alias scheme to use only the probability table $F$, and retry the sample in the case that in the original would output an entry from $A$. Because the known bounds are defined to be constant, the rejection sampling takes expected constant time, and updates only require a change to the single entry of $F$ that has been modified. Optimizations involve fine-grained upper bounds for each individual entry of the vector, and techniques to correct for incorrect upper bound estimates, but the spirit of the algorithm remains the same.

### 2.2.3 Tree-Based Methods

An alternative approach for sampling from dynamic probability distributions originates in Wong and Easton [20]. Each count in the vector comprises the leaf of a tree, and the intermediate nodes contain the sum of the values of their children; samples are performed by uniformly drawing a number less than the vector's total, and traversing the tree to find the relevant bucket. Therefore, the costs of precomputation, samples, and updates correspond exactly to construction, lookups, and insertions in a self-balancing binary search tree (or a skip list, or whatever other equivalent structure is used). If we fix $k$ and only allow updates to change the frequency of each element, we can use a binary tree without rebalancing operations, because each update only changes the metadata associated with each internal node, not the tree structure. Therefore, this technique achieves updates in $O(\log k)$ time,

| Algorithm | Setup time | Sample time | Update time | Assumptions |
|---|---|---|---|---|
| Vose [18] | $\Theta(k)$ | $\Theta(1)$ | $\Theta(k)$ | sufficient precision |
| RR [14] | $\Theta(k)$ | $O_{\mathbb{E}}(1)$ | $\Theta(1)$ | *a priori* bounds |
| WE [20] | $\Theta(k)$ | $\Theta(\log k)$ | $\Theta(\log k)$ | none |
| MVN1 [10] | $O(k)$ | $O_{\mathbb{E}}(\log^* k)$ | $O_{A\mathbb{E}}(\log^* k)$ | O(1) integer log |
| HMM [4] | $O(k)$ | $O_{\mathbb{E}}(1)$ | $\Theta(1)$ | poly(k) bound |
| MVN2 [10] | $O(k)$ | $O_{\mathbb{E}}(1)$ | $O_{\mathbb{E}}(1)$ | O(1) integer log |

Figure 1: Asymptotic comparison of exact categorical sampling algorithms.

improved over the alias method, but sample performance degrades to $O(\log k)$.

Matias et al. [10] use a broadly tree-based approach to achieve sampling in $O_{\mathbb{E}}(\log^* k)$ time and updates in $O_{A\mathbb{E}}(\log^* k)$ time. Rather than construct a tree of depth $\log k$, they construct a forest of trees with depth at most $\log^* k$. Each level $\ell$ consists of $\log w$ buckets $R_j^{(\ell)}$ filled from left to right representing the ranges $[2^{j-1}, 2^j)$, where $w$ is both the word size and the total of all counts in the probability vector. If a bucket contains more than one element, this process repeats hierarchically at the next level, otherwise this root is added to the dynamically sized array for this level of the tree, which also keeps track of the total count among all its rooted trees.

Sampling from this data structure requires three steps. First, generate a uniformly random number in $w$ and find the lowest level of the tree whose roots cover this number; this clearly takes $O(\log^* k)$ time. Second, choose the minimum nonempty root range on that level that covers the random number by iterating sequentially over them; this may be as many as $\log w$ entries, but the expected cost is constant. Third, use rejection sampling to choose a child node at each level; this again has constant expected cost, and expected $O_{\mathbb{E}}(\log^* k)$ across all levels.

The basic approach to update and rebalance this structure takes $O(2^{\log^* k})$ time, but weakening the range guarantee for each bucket from $[2^{j-1}, 2^j)$ to $[(1-b)2^{j-1}, (2+b)2^{j-1})$ for some $0 \leq b < 1$ allows for lazy updates and the stated amortized cost of $O_A(\log^* k)$. This comes at the cost of additional machinery in the sampling algorithm, but the asymptotic complexity remains the same.

Hagerup et al. [4] take a superficially different approach that results in essentially an improved version of the same algorithm. Here the solution is posed as a two-level recursive reduction which expands into $\log w$ instances of subdistributions with particular structure at each step, then collapses into a table lookup once the instances are small enough for enumeration to be feasible. Here there is the additional constraint that the entries of the vector may only be polynomially large in $k$.

Matias et al. apply the same optimization to all but two levels of their tree, and achieve roughly the same performance. While the details of their scheme manage to avoid the additional constraints of Hagerup et al., this comes at the cost of expected rather than worst-case constant update time.

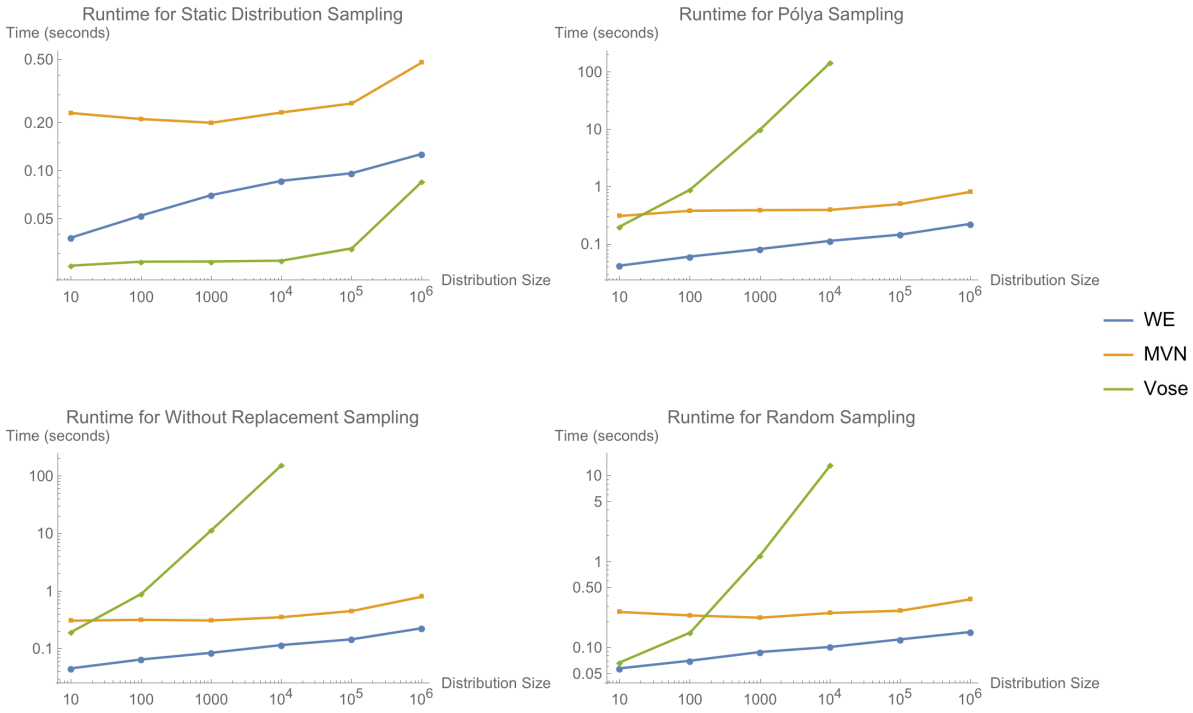


Figure 2: Runtime vs. distribution size of categorical sampling algorithms. Both axes are log scale.

## 2.3 Evaluation

In order to validate the real-world utility of these methods, we have implemented the algorithms of Wong and Easton (WE), Matias et al. (MVN), and Vose (Vose) in C++ and optimized the implementations to avoid any excessive dynamic allocation, poor cache utilization, or other bottlenecks revealed during profiling. The WE algorithm was implemented using a flattened tree to improve caching behavior. The MVN algorithm was implemented using the C++ STL's hash table, `unordered_map`. Furthermore, we selected the first MVN algorithm (MVN1) for our implementation, as it was the more clearly specified in the paper and did not rely on heuristic reorganizations. The original paper omitted many of the details for the tree data structure used to achieve constant time reorganization operations, but we achieved the same result using augmented back-pointers from children in the tree. To our knowledge, this is the first concrete implementation of MVN. Our implementation of Vose followed the algorithm exactly, but we organized the alias table queues to improve cache efficiency.

To evaluate the algorithms, we tested four sampling patterns:

- Static Distribution Sampling: In this pattern, we simply construct a random distribution of size $k$ and take $n = 10^6$ random samples. The distribution is constructed by assigning weights from the uniform integer distribution on $[0, m)$. No updates are performed.

- Pólya Sampling: In this pattern, we construct a uniform categorical distribution of size $k$ with densities $p_i = 1$. Each time we draw an element $i$, we update $p_i \leftarrow p_i + 1$. We perform a total of $n = 10^6$ draws.

- Without Replacement Sampling: In this pattern, we construct a uniform categorical distribution of size $k$ with densities $p_i = n/k$. Each time we draw an element $i$, we update $p_i \leftarrow p_i - 1$. We perform a total of $n = 10^6$ draws.

- Random Sampling: In this pattern, we construct a random distribution of size $k$ and perform $n = 10^6$ operations. Each of these operations is a random update with probability $q$ and a random sample with probability $1 - q$. In our evaluation, we used $q = 0.1$.

The benchmarks were run on a 6-core, 12-thread machine with an Intel i7-3930k CPU at 3.8 GHz and 32 GB of DDR3 RAM. The machine ran Ubuntu 18.04 on Linux 4.15.0-39. Compilation was performed with g++ version 7.3.0-27 and the options "`-std=c++11 -O3`".

Our Static Distribution results reveal the behavior of the three algorithms in the absence of updates. Vose, which draws samples in $O(1)$ time, is considerably faster than the other two methods at all sample sizes. Surprisingly, the runtime of Vose increases significantly when the distribution size reaches $10^6$. This is because the construction time of the alias table begins to dominate the test; when the alias table grows large, the construction routine involves a significant number of nearly-random memory accesses across a multiple-megabyte table. This interacts with the thresholds between levels of the cache hierarchy to cause a sudden and disproportionate increase in the constant factor overhead of the construction phase, dwarfing the time required for samples on the precomputed table. WE behaves as expected, increasing roughly logarithmically with the size of the distribution. The behavior of MVN is far more erratic. Initially, as the size of the distribution increases, MVN's runtime remains the same or decreases slightly. The cause of this appears to be the tree construction; larger distribution sizes can spread the distribution across a larger number of shallower trees. Unexpectedly, profiling reveals that the majority of time is spent in the rejection sampling portion of the algorithm, which is strained when the number of trees is low. As the distribution size increases, we find that most CPU time is spent in rejection sampling and hash table lookups.

Pólya Sampling, Without Replacement Sampling, and Random Sampling all yield similar results. When regular updates are introduced, MVN and WE begin to outperform Vose at very small distribution sizes. This is because with every update, the Vose algorithm must recompute the entire alias table, consuming $O(k)$ time, where $k$ is the size of the distribution. For WE, this takes only $O(\log k)$ time, and for MVN, this takes only $O_{\mathbb{E}}(\log^* k)$ time. As such, Vose sampling becomes infeasible above a distribution size of 1000, while both MVN and WE remain viable through the upper limits of our testing (a distribution size of $10^6$). Though MVN appears to approach the runtime of WE, we find that as the distribution size is increased further, memory management begins to dominate the runtime of MVN due to its dynamic tree structure (over 10GB are used when $k = 10^8$). This makes the algorithm infeasible for distributions of size $k = 10^9$ and larger, while WE remains viable.

# 3 Multinomial Sampling

## 3.1 Background

The problem of *multinomial sampling* can be defined in terms of categorical sampling as follows: given the categorical distribution defined by a vector $a \in \mathbb{N}^k$, we draw $n$ samples and output a vector $b$ with the frequency of draws in each category, where $\sum_{i=0}^{k} b_i = n$. The multinomial distribution is a generalization of the better-known binomial distribution, in which we draw repeatedly from a Bernoulli distribution.

## 3.2 Existing Algorithms

Underlying most pre-existing work on efficient multinomial sampling are efficient binomial sampling algorithms which exploit properties of the marginal and conditional multinomial distribution. Specifically, given a multinomial distribution with $k$ categories with associated probabilities $p_1, p_2, \ldots, p_k$ and count $n$, we can compute a sample $(c_1, c_2, \ldots, c_k)$ by leveraging the fact that $c_i \sim \text{Binom}(p_i, n)$. Furthermore, conditioning on the values we have already obtained, we have that $c_j | c_{i_0}, c_{i_1}, \ldots, c_{i_m} \sim \text{Binom}\left(\frac{p_j}{1-p_{i_0}-p_{i_1}-\cdots-p_{i_m}}, n - c_{i_0} - c_{i-1} - \cdots - c_{i_m}\right)$. This decomposition allows us to write efficient multinomial sampling algorithms in terms of efficient binomial sampling algorithms.

### 3.2.1 Inverse CDF Sampling

The inverse transformation is a simple method for efficient binomial sampling summarized in [9]. In general, the inverse transformation refers to sampling via the inverse CDF, a methodology referred to by other sources as the probability integral transform or universality of the uniform. Its correctness follows from the result that given a distribution $X$ with CDF $F_X$, the random variable $F_X^{-1}(U) \sim X$ for $U \sim \text{Unif}(0, 1)$.

Fortunately, in the case of a binomially distributed $X \sim \text{Binom}(p, n)$, $F_X^{-1}(x)$ can be easily computed by summing the $n$ values of the PDF $f_X(k)$ until the cumulative sum exceeds $x$. As $f_X$ has a simple recursive form, each of these PDF values can be computed in $\Theta(1)$ time. Thus sampling takes $O_{\mathbb{E}}(np)$ time per the expected value of the binomial distribution. Utilizing this sampling methodology for a multinomial distribution, we obtain the runtime $O_{\mathbb{E}}(n)$.

### 3.2.2 Rejection Sampling Methods

The most efficient variants of binomial sampling algorithms make use of acceptance/rejection sampling, also mentioned in Section 2.2.2. In essence, these algorithms use a majorizing and minorizing function, which are strictly greater than or less than the PDF of the distribution respectively. The resulting region under the majorizing function can then be subdivided into regions such that samples are unconditionally accepted, unconditionally rejected, or are accepted based on a sample from an easily computable distribution.

BTPE, as described in [9], uses this methodology using a linear absolute value function as its minorizing function and a piecewise linear and exponential function as its majorizing function. This splits the region into four logical sub-regions that each allow for simple acceptance-rejection testing. The authors show that $O(1)$ uniform random variables are required in the average case, provided a sufficiently large word size.

The BRTD algorithm described in [5] uses a very similar methodology, with similar majorizing and minorizing functions, along with a clever scheme for recycling random variates when the algorithm decides to reject a sample. This allows us to bring the expected number of random variates (of sufficient precision) down to about 2.5.

### 3.2.3 Alternative Representations

Other efficient algorithms for binomial generation rely on alternative representations of binomial random variables. The most straightforward of these is the Bernoulli representation, in which a binomial $X \sim \text{Binom}(p, n)$ is represented as $X = B_1 + B_2 + \cdots + B_n$ for $B_i \overset{i.i.d.}{\sim} Bern(p)$. As Bernoulli random variables can be efficiently generated by sampling a uniform distribution and comparing the result to $p$, this yields a very straightforward $O(n)$ sampling algorithm. An algorithm with comparable runtime to the inverse transformation approach can be produced using a representation in terms of a geometric distribution [9].

### 3.2.4 The Relles Algorithm

The Bernoulli representation yields another natural algorithm proposed in [15] using binary search (referred to henceforth as the Relles algorithm). Note that to generate $n$ independent Bernoulli samples, we can generate $n$ independent uniform samples and determine how many fall below the probability $p$. As each of the uniform samples is independent and below $p$ with probability $p$, we thus obtain the proportion of Bernoulli samples that are 1 (falling below $p$) and 0 (falling above $p$). These counts can then generate our binomial random variable.

A naive implementation would require $O(n)$ time to generate the $n$ uniform samples, but Relles proposes to instead sort the list of uniform samples and binary search for the boundary point at probability $p$. When this point's index is found, we then immediately have the number of samples falling below $p$. Again, sorting appears to be infeasible as it would introduce a $O(n \log n)$ overhead. However, note that as we are traversing the sorted array via binary search, we ultimately only need to determine the values of $O(\log n)$ uniform samples, located at the indices we examine. These uniform samples can be generated directly using the representation of uniform order statistics in terms of beta random variables.

Suppose we have a sorted array of independent uniform random variables on $[0, 1)$, which we denote $U_{(1)}, U_{(2)}, \ldots, U_{(m)}$. Define a set of $n + 1$ i.i.d. exponential random variables $X_1, X_2, \ldots, X_{n+1}$. We have that $U_{(i)} = \frac{X_1 + \cdots + X_i}{X_1 + \cdots + X_{n+1}}$. Thus, we have that marginally $U_{(i)} \sim \text{Beta}(i, n - i + 1)$, and given that $b - a + 1$ order statistics lie within the interval $[U_{(a)}, U_{(b)})$ (in which no other points are known), $U_{(a+c)} \sim U_{(a)} + (U_{(b)} - U_{(a)})B$ where $B \sim \text{Beta}(c, b - a + 1)$ for $a + c < b$. This thus provides us with a method of drawing uniform order statistics given

known endpoints [15]. By generating these samples as our binary search proceeds, the Relles algorithm terminates in $O(\log n)$ time, yielding a binomial sample.

We can naturally extend this approach to multinomial sampling with categorical probabilities $(p_1, p_2, \ldots, p_k)$ and total count $n$ by again considering an unmaterialized sorted array of $n$ random variates. To generate the sample $(y_1, y_2, \ldots y_k)$, we first perform binary search for $p_1$. The index obtained is our value for $y_1$. We then binary search for $p_1 + p_2$ to obtain the value for $y_1 + y_2$, from which we can obtain $y_2$. We repeat this process, determining $\sum_{i=1}^{m} y_i$ by binary searching from $\sum_{i=1}^{m} p_i$. However, note that each of these $n$ searches are dependent and must consider the uniform variates that were generated in previous rounds. Thus we store each of the previously computed values in a hash table for retrieval. Whenever we wish to obtain the value of the uniform random variate at an index, we check the hash table for a precomputed value. If this value is absent, then we interpolate from the necessarily known upper and lower bounds. Thus the algorithm achieves runtime $O(k \log n)$.

## 3.3 More Efficient Beta-based Generation

Our main theoretical contribution is to improve the runtime of Beta-based multinomial sampling from $O(k \log n)$ using the Relles algorithm to $O(k \log \log n)$ using our related searching algorithm. Specifically, we substitute interpolation search for binary search and demonstrate that this reduces the search time from $O(\log n)$ to $O(\log \log n)$. We further show how to conserve random variates in this scheme by leveraging precomputed values.

### 3.3.1 Interpolation Search

Interpolation search [13] is a search algorithm for locating a value in a sorted array. Similarly to binary search, interpolation search compares the target with a value in the array to subdivide the remaining array into two portions. If the target value matches the value in the array, we have found our match. If our target is smaller, we restrict our search to the left portion. If it is larger, we restrict our search to the right portion. The difference between binary and interpolation search lies in their selection of the subdivision point in the array. For binary search, this is always chosen to be the center. Thus the interval in which our value lies shrinks by a factor of 2 on each iteration, yielding a runtime of $O(\log n)$.

Interpolation search instead tracks the endpoints of the current interval and assumes that all values within the interval are uniformly distributed between these two endpoints. As a result, it interpolates between these bounds to find the expected position of the target value, assuming the points are uniformly distributed. This is a poor heuristic for arrays that follow more complex distributions, but for arrays that are truly uniformly distributed, it can easily be shown that the expected runtime is $O_{\mathbb{E}}(\log \log n)$. See [12] for an overview of this proof.

Fortunately, the array under consideration in the Relles algorithm is generated by sorting $n$ uniform samples on the range $[0, 1)$. This is exactly the distribution assumed by interpolation search. Thus, by locating our uniform order statistic via interpolation search instead of binary search, we improve the runtime of the Relles algorithm from $O(k \log n)$ to $O_{\mathbb{E}}(k \log \log n)$.

### 3.3.2 Generalizing to Multinomial

While this yields a straightforward $O_{\mathbb{E}}(\log \log n)$ variant of a binomial sampling algorithm, there are two paths to achieving $O_{\mathbb{E}}(k \log \log n)$ multinomial sampling. The simpler solution uses the conditional representation of each portion of the multinomial sample's output as a binomial random variable (see 3.2). Therefore we perform $k$ binomial samples to generate our multinomial sample.

Alternatively, we can adapt our extension to perform interpolation search on the same array, caching previously computed values. Note that this as not as straightforward as memoization in the binary search variant. With binary search, because we always traverse the array through subdivision, whenever we need to compute a new value the endpoints of the current interval are guaranteed to contain no other known values. This is important, as the distribution of the current point is Beta-distributed given these two endpoints if no other points within the interval are known. Therefore, to extend this procedure to interpolation search we must track our endpoints at every iteration of the search.

This is trivial to do within a single search, but we must also maintain these endpoints between the $k$ search operations. To do so, we maintain a stack that persists across each of the searches. The stack initially contains only the pair $(n, 1)$, indicating that the value of the $n$th order statistic is fixed at 1. When searching for a value $p$, if a computed order statistic $p_i$ at index $i$ exceeds $p$, we push $(i, p_i)$ onto the stack. As this value exceeds $p$, any values encountered later during the search will be strictly less than $p_i$. Therefore, the stack is sorted in ascending pop-order.

When we begin our search for the next probability $p'$, we must locate the known endpoints within which our target point lies. To do so, we pop values off of the stack until we find one that exceeds $p'$. This point becomes our upper bound and is returend to the stack, and the previously popped value (or $(0, 0)$ if there was no previous value) becomes our lower bound. Note that because the previous target value $p$ could not be located without finding two adjacent values smaller and larger than it respectively, the lower bound must be the largest value in the array less than $p$, or a larger value. Thus our interval is non-overlapping with previous points (or empty). Furthermore, no points within our interval are known, as our endpoints were adjacent on the ordered stack and thus any points between them would have been pushed. Known points within this interval could not have been consumed by a previous search's popping operations, as the upper bound was returned to the stack and any points produced by the subsequent traversal that could be within a later point's search region were pushed. Thus the stack maintains correctness across search operations.

Furthermore, note that traversing the stack does not affect the runtime of the algorithm. By using interpolation search, we expect a total of $O_{\mathbb{E}}(k \log \log n)$ points to be computed over the course of the sample generation. Therefore, at most $O_{\mathbb{E}}(k \log \log n)$ points will be pushed onto the stack. Each point is popped off of the stack exactly once, except for the $O(k)$ occasions on which a point is returned to the stack. Therefore, we spend at most $O_{\mathbb{E}}(k \log \log n)$ time traversing the stack, maintaining our original runtime.

| Algorithm | Setup time | Sample time | Update time | Assumptions |
|---|---|---|---|---|
| Vose [18] | $\Theta(k)$ | $\Theta(\max(k, n))$ | $\Theta(k)$ | sufficient precision |
| Relles [15] | $\Theta(1)$ | $\Theta(k \log n)$ | $\Theta(1)$ | efficient Beta r.v. |
| This work | $\Theta(1)$ | $\Theta_{\mathbb{E}}(k \log \log n)$ | $\Theta(1)$ | efficient Beta r.v. |
| BTPE [9] | $\Theta(1)$ | $\Theta_{\mathbb{E}}(k)$ | $\Theta(1)$ | sufficient precision |
| BRTD [5] | $\Theta(1)$ | $\Theta_{\mathbb{E}}(k)$ | $\Theta(1)$ | sufficient precision |

Figure 3: Asymptotic comparison of exact multinomial sampling algorithms.

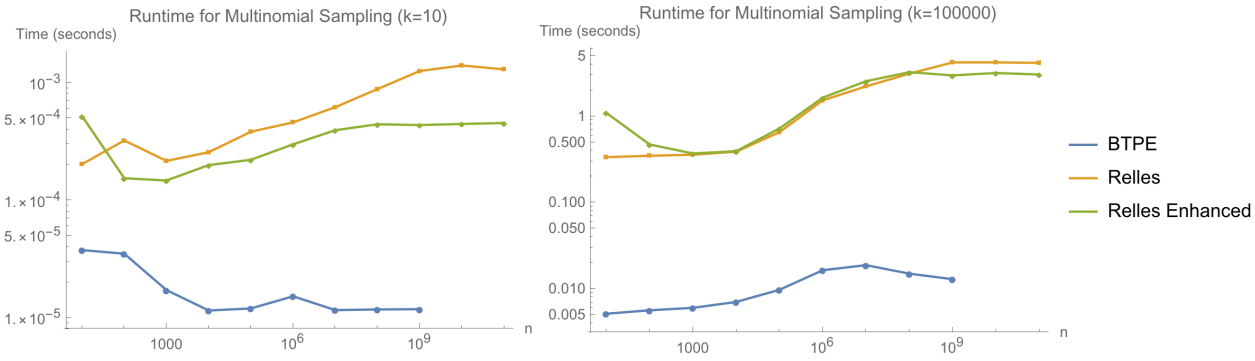


Figure 4: Runtime vs. $n$ of multinomial sampling algorithms, across two choices of $k$. Both axes are log scale.

## 3.4 Evaluation

To assess our enhanced version of the Relles algorithm, we performed a benchmark of the two Relles variants and the GNU Scientific Library's BTPE implementation. The Relles implementations tested are our original implementations, using C++ STL's `unordered_map` to store uniform order statistics.

The benchmark first generates a multinomial distribution of size $k$ with random uniform weights and rescales the weights such that they sum to 1. It then performs a multinomial sample of size $n$. A total of 5 trials are run for each combination of $n$, $k$, and algorithm. The three algorithms were tested for two distribution sizes, $k = 10$ and $k = 10^5$.

The benchmarks were run on a 6-core, 12-thread machine with an Intel i7-3930k CPU at 3.8 GHz and 32 GB of DDR3 RAM. The machine ran Ubuntu 18.04 on Linux 4.15.0-39. Compilation was performed with g++ version 7.3.0-27 and the options "`-std=c++11 -O3 -lgsl -lgslcblas`".

The BTPE implementation performed significantly better than either Relles variant, with runtime remaining mostly constant despite increasing $n$ (as expected). The GNU Scientific Library implementation of BTPE does not currently support a word size larger than 32-bits, so we did not evaluate BTPE algorithm above this threshold.

At moderate sample sizes, we noticed an improvement of the interpolation search variant of Relles over the original. With small $k$ and large $n$, we observed improvements of up to

65%. With large $k$ and large $n$, improvements reached about 26%. For large $k$ and small $n$, interpolation search offers less of an advantage, as little time is spent binary searching through the unmaterialized uniform order statistics.

# 4 Approximate $\ell_1$ Sampling

## 4.1 Background

The problem of *approximate $\ell_1$ sampling*, or *approximate categorical sampling*, is to draw a single sample $x$ according to the rule $\Pr[x = i] = (1 \pm \epsilon)\frac{a_i}{\|a\|_1} + O(k^{-c})$, with a $\delta$ probability of total failure, using a sketch of size $o(k)$. The question of whether this is possible was first posed in [2], then later in [3], and finally answered positively in [11].

There is a known $\Omega(\log^2(k)\log(\frac{1}{\delta}))$ space lower bound for this problem in [8]; this bound's lack of dependence on $\epsilon$ motivates the framing of a related problem. In *perfect $\ell_1$ sampling*, we draw a single sample $x$ according to the rule $\Pr[x = i] = \frac{a_i}{\|a\|_1} + O(k^{-c})$, with a $\delta$ probability of total failure, using a sketch of size $o(k)$. That is to say, perfect $\ell_1$ sampling is a strengthened version of approximate $\ell_1$ sampling that eliminates relative error.

The literature on this subject decomposes algorithms into several phases: in the processing phase, the streaming updates to our implicit vector $a$ are phrased as a linear sketch $L : \mathbb{R}^k \to \mathbb{R}^{O(S \log k)}$ applied to the entire input vector at once. However, because the sketch is defined to be linear, we can represent an update $(i, c)$ as $L(a + ce_i) = La + cLe_i$, where $e_i$ is the one-hot vector encoding of $i$. The randomized values in this linear function $L$ are often hidden away in the random oracle model; we might simulate this in practice with a pseudorandom function family, though this introduces standard cryptographic assumptions to maintain correctness and complicates implementation. Sometimes $L$ is explicitly derandomized, at the cost of marginally higher space complexity. The time complexity of this phase is usually analyzed. In the recovery stage, we are given $L$ and $a$ and wish to draw an $\ell_1$ sample. Often the time complexity of this phase falls by the wayside, and algorithms are described more as information-theoretically extracting information from a sketch than as an efficient procedure to do so. In particular, it is often the case that these algorithms iterate over the entire vector of length $k$, which is not suited to real streaming applications.

Note that here our use of the strict turnstile model is significant. For the cash register model, in which updates $c$ must be strictly positive, $\ell_1$ sampling can be solved exactly in constant space with constant time updates using a simple reservoir sampling algorithm [17]: store the first item in memory, and for each $n$th additional item, replace the stored item with probability $\frac{1}{n}$. We see that this simple algorithm is fundamentally incompatible with the ability to subtract counts, and so in some sense subtraction is central to the difficulty of $\ell_p$ sampling.

## 4.2 Algorithms

The body of work on $\ell_1$ sampling is fairly homogeneous, following a paradigm called *precision sampling* introduced by Andoni et al. [1]. Suppose we are trying to estimate the sum of elements in $a \in \mathbb{R}^k$, where $a_i \in [0, 1]$. In contrast to standard Monte Carlo sampling, in which a random subset of elements are selected, their values summed, and the total rescaled based on the size of the sample, precision sampling chooses a sequence of precisions $u_i$ independently of $a$ and finds estimates $\hat{a}_i$ such that $|\hat{a}_i - a_i| \leq u_i$, for all $i$. The idea is generally that $a_i$'s estimate should be the maximum over $a$ with probability (w.r.t. randomness in the choice of $u$) roughly proportional to $a_i$.

Enhancements to the algorithms come mostly from small modifications and improved bounds rather than fundamentally new thinking or data structures. In fact, virtually all $\ell_1$ sampling algorithms are built atop the basic *count sketch*. In brief, a count sketch is a set of $J$ hash tables of size $m$, each of which has random hash functions $h_j : [k] \to [m]$ and $g_j : [k] \to \{-1, 1\}$. When an update $(i, c)$ occurs, for each hash table $j$ we index into cell $h_j(i)$ and add the value $g_j(i) \cdot cw_i$, where $w$ are the precision weights associated with the linear sketch $L$. (Traditionally, a number of weights are drawn from $\text{Unif}(0, 1)$ for each $i$, and the maximum of these draws is set to $w_i$.) In the context of $\ell_p$ sampling, $J$ is usually $O(\log k)$.

At a high level, the variations on this template are as follows: the original precision sampling algorithm by Andoni et al. [1] computes the median $\hat{a}_i$ across hash tables $j$ and tries to find a hash table whose weights make only one index's median estimate cross a particular precision threshold. The algorithm as described uses space $O(\frac{1}{\epsilon}\log^3(k)\log(\frac{1}{\delta}))$, performs updates in time $O(\frac{1}{\epsilon}\log(k))$, and draws a sample in time $O(k\log(k))$. Elsewhere in the paper, the authors describe how related algorithms can have their runtime improved using a heavy-hitters sketch subroutine and an additional $\log\log$ space factor, but there is no such analysis extending these results to $\ell_1$ sampling.

Jowhari et al. [7] use largely the same format, but maintain an additional sketch to track the $\ell_2$ norm of the precision-scaled entries. Using this, they find a tighter bound on the error of the count sketch and so can instantiate it with smaller space parameters, because the statistical test that detects the algorithm's failure with probability $\delta$ is closer to ideal. This reduces the space complexity to only $O(\frac{1}{\epsilon}\log(\frac{1}{\epsilon})\log^2(k)\log(\frac{1}{\delta}))$. However, this comes at the cost of Andoni et al.'s reliance on only *pairwise* independence in the generation of the random precision weights. There is no analysis of time complexity in this paper.

Jayaram et al. [6] use this template to instantiate a perfect $\ell_1$ sampler, rather than just an approximate one. Their key insight is to generate the precision weights not with a maximum over a batch of uniform samples, but with samples from the exponential distribution. The analysis following this manages to meet the aforementioned lower bound in the random oracle model, and come fairly close in the derandomized setting, using space $O(\log^2(k)(\log\log k)^2\log(\frac{1}{\delta}))$. The paper proves that updates and samples can be performed in time $polylog(k)$, an improvement in sample time over the naive schemes of previous papers.

| Algorithm | Space | Sample time | Update time | Type |
|---|---|---|---|---|
| AKO [1] | $O(\frac{1}{\epsilon}\log^3(k)\log(\frac{1}{\delta}))$ | $O(k\log k)$ | $O(\frac{1}{\epsilon}\log k)$ | approx |
| JST [7] | $O(\frac{1}{\epsilon}\log(\frac{1}{\epsilon})\log^2(k)\log(\frac{1}{\delta}))$ | $O(k\log k)$ | variable | approx |
| JW [6] | $O(\log^2(k)(\log\log k)^2\log(\frac{1}{\delta}))$ | $polylog(k)$ | $polylog(k)$ | perfect |

Figure 5: Asymptotic comparison of approximate categorical sampling algorithms.

# 5 Approximate Multinomial Sampling

It is natural to ask whether there is an extension of multinomial sampling to the streaming setting, analogous to the transformation of categorical sampling into $\ell_1$ sampling, but this setting is entirely absent from the literature. We will briefly discuss why the general problem of approximate multinomial sampling is at best uninteresting and at worst ill-posed.

We might reasonably define *approximate multinomial sampling* as follows: given a stream of updates to the implicit vector $a \in \mathbb{N}^k$, we wish to draw $n$ independent successful $\ell_1$ samples (with relative error $\epsilon$) and output the frequency of each of $k$ categories in a vector $b \in \mathbb{R}^k$. However, by defining the output as a vector of all categories, as in the exact case, we necessitate $\Theta(k)$ space usage. We could alleviate the concrete effect of this by streaming the output rather than writing it down contiguously, but this still requires our sampling algorithm to take $\Omega(k)$ time, which is impractical.

A simple but drastic way to resolve this would be to change the output format to include only 1 specified category. This reduces the problem to finding a point estimate $p_i$ for the probability of each element, and then performing an approximate binomial sample on the particular desired $p_i$ and $1 - p_i$. But this can be trivially computed using any sketch that provides unbiased point query estimates, and so is not truly a distinct problem.

A potentially interesting middle ground would be to output only those categories that have non-zero counts, which for a sparse universe would ideally make output sizes tractable. Here we run into the motivation for additive error in the definition of $\ell_1$ sampling, which is even easier to illustrate in the context of approximate multinomial sampling. We can show by reduction to the augmented index problem that any sketching algorithm for multinomial sampling that has only relative error (meaning that the probability of any element whose true probability is 0 will be approximated as 0) must use $\Omega(k)$ space.

The augmented index problem from communication complexity is defined as follows: Alice has a string $x \in \{0,1\}^k$, and Bob has an index $i \in [k]$. When messages can only flow from Alice to Bob, $\Omega(k)$ communication is required for Bob to learn $x_i$ with probability at least $\frac{2}{3}$. We now outline a sketch of the proof: suppose for the sake of contradiction that there exists a sublinear space algorithm for approximate multinomial sampling without additive error. Alice adds 1 count to $a_i$ for each $i$ such that $x_i = 1$, and sends the sketch to Bob. Bob performs a multinomial sample for sufficiently large $n$ ($O(k/(1-\epsilon))$) such that the probability of not having observed an element with nonzero probability is a negligible constant by asymptotic bounds on the binomial pmf. This allows Bob to recover the entire string, and so the $x_i$ of his choice, which is a contradiction.

Since the output will not remain sparse as we approach $n = \Omega(k)$, we might finally restrict the multinomial sampling problem to $n = o(k)$, and ask whether this can be done more efficiently than the naive $n$ independent instantiations of $\ell_1$ sampling, which already uses $o(k)$ space. For particularly nonuniform distributions these results might be useful, but this is a much narrower case than the original multinomial sampling problem and does not have clear motivation.

# 6 Conclusion

In this paper, we presented a survey of algorithms for exact categorical sampling, exact multinomial sampling, and approximate categorical sampling, and provided some intuition into why there is no such field studying approximate multinomial sampling. Our C++ implementations of exact categorical and multinomial sampling algorithms and the testing apparatus we used to perform our evaluation are publicly available at sampling.

With respect to exact categorical sampling, we demonstrated that despite the asymptotic gains of Matias et al.'s never-before-implemented state-of-the-art categorical sampling algorithm [10], in practice Wong and Easton's algorithm [20] is preferable for dynamic sampling, and Vose's [18] for static sampling.

In the context of exact multinomial sampling, we improved the best beta-based multinomial sampling algorithm [15] from $O(k \log n)$ to $O(k \log \log n)$, which results in a substantial real-world performance improvement that admittedly still pales in comparison to the $O(k)$ approach of Kachitvichyanukul and Schmeiser [9].

We also presented a unified summary of techniques in approximate categorical sampling, or $\ell_1$ sampling, and in particular described the precision sampling paradigm, which is followed by most recent $\ell_1$ sampling algorithms. And finally, we discussed how the apparent lacuna around approximate multinomial sampling in the literature may be attributed to subtle problems in the formulation of its definition.

As we have reached the asymptotic lower bounds in complexity for all these problems, the most relevant direction for future work is to devise and implement optimal algorithms with lower constant factor overhead. In particular, Matias et al. [10] describes several high-level optimizations to improve asymptotic performance over the version we implemented, but many crucial details that determine real performance are absent. Meanwhile, the $\ell_1$ sampling algorithms culminating in Jayaram et al. [6] rely on models and assumptions that do not translate directly to actual computers. An evaluation of $\ell_1$ sampling in some application where reservoir sampling and strictly positive updates do not suffice would greatly strengthen the motivation for the problem, which is still somewhat nebulous compared to that of exact sampling methods. As we have seen in experiments across categorical and multinomial sampling methods, standard binary-tree based methods and simple rejection sampling often outperform more clever theoretical constructs at realistic input scales.

# References


[1] Andoni, Alexandr, et al. "Streaming Algorithms via Precision Sampling." *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, 2011.

[2] Cormode, Graham, et al. "Summarizing and Mining Inverse Distributions on Data Streams via Dynamic Inverse Sampling." *Proceedings of the 31st VLDB Conference*, 2005.

[3] Frahling, Gereon, et al. "Sampling in Dynamic Data Streams and Applications." *Proceedings of the Twenty-First Annual Symposium on Computational Geometry - SCG '05*, 2005.

[4] Hagerup, T., K. Mehlhorn, and I. Munro, "Optimal Algorithms for Generating Discrete Random Variables with Changing Distributions," Max Planck Institute, Technical Report MPI-I-92-145, October 15, 1992.

[5] Hörmann, Wolfgang. "The Generation of Binomial Random Variates." *Journal of Statistical Computation and Simulation*, vol. 46, no. 1-2, 1993, pp. 101–110.

[6] Jayaram, Rajesh, and David P. Woodruff. "Perfect $L_p$ Sampling in a Data Stream." *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, 2018.

[7] Jowhari, Hossein, et al. "Tight Bounds for $L_p$ Samplers, Finding Duplicates in Streams, and Related Problems." *Proceedings of the 30th Symposium on Principles of Database Systems of Data - PODS '11*, 2011.

[8] Kapralov, Michael, et al. "Optimal Lower Bounds for Universal Relation, and for Samplers and Finding Duplicates in Streams." *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, 2017.

[9] Kachitvichyanukul, Voratas, and Bruce W. Schmeiser. "Binomial Random Variate Generation." *Communications of the ACM*, vol. 31, no. 2, 1988, pp. 216–222.

[10] Matias, Yossi, et al. "Dynamic Generation of Discrete Random Variates." *Theory of Computing Systems*, vol. 36, no. 4, 2003, pp. 329–358.

[11] Monemizadeh, Morteza, and David P. Woodruff. "1-Pass Relative-Error $L_p$-Sampling with Applications." *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, 2010, pp. 1143–1160.

[12] Perl, Yehoshua, and Edward M. Reingold. "Understanding the Complexity of Interpolation Search." *Information Processing Letters*, vol. 6, no. 6, 1977, pp. 219–222.

[13] Peterson, W. W. "Addressing for Random-Access Storage." *IBM Journal of Research and Development*, vol. 1, no. 2, 1957, pp. 130–146.

[14] Rajasekaran, Sanguthevar, and Keith W. Ross. "Fast Algorithms for Generating Discrete Random Variates with Changing Distributions." *ACM Transactions on Modeling and Computer Simulation*, vol. 3, no. 1, 1993, pp. 1–19.

[15] Relles, Daniel A. "A Simple Algorithm for Generating Binomial Random Variables When N Is Large." Journal of the American Statistical Association, vol. 67, no. 339, 1972, pp. 612–613.

[16] Schwartz, Keith. *Darts, Dice, and Coins: Sampling from a Discrete Distribution.* 2011, www.keithschwarz.com/darts-dice-coins/.

[17] Vitter, Jeffrey S. "Random Sampling with a Reservoir." *ACM Transactions on Mathematical Software*, vol. 11, no. 1, 1985, pp. 37–57.

[18] Vose, Michael. "A Linear Algorithm for Generating Random Numbers with a given Distribution." *IEEE Transactions on Software Engineering*, vol. 17, no. 9, 1991, pp. 972–975.

[19] Walker, Alastair J. "An Efficient Method for Generating Discrete Random Variables with General Distributions." *ACM Transactions on Mathematical Software*, vol. 3, no. 3, 1977, pp. 253–256.

[20] Wong, C. K., and M. C. Easton. "An Efficient Method for Weighted Sampling without Replacement." SIAM Journal on Computing, vol. 9, no. 1, 1980, pp. 111–113.