Fast Perfect Weighted Resampling

Bart Massey
Assoc. Prof. Computer Science
Portland State University
bart@cs.pdx.edu

Draft of October 20, 2007
Do Not Distribute

Abstract

We describe an algorithm for perfect weighted-random resampling of a population with time complexity $O(m + n)$ for resampling m inputs to produce n outputs. This algorithm is an incremental improvement over standard resampling algorithms. Our resampling algorithm is parallelizable, with linear speedup. Linear-time resampling yields notable performance improvements in our motivating example of Sequential Importance Resampling for Bayesian Particle Filtering.

1 Introduction

Bayesian Particle Filtering (BPF) [**?**] is an exciting new methodology for state space tracking and sensor fusion. The bottleneck step in a typical BPF implementation is the weighted resampling step known as Sequential Importance Resampling: creating a new population of "particles" from an old population by random sampling from the source population according to the particle weights. Consider resampling m inputs to produce n outputs. The naïve algorithm has time complexity $O(mn)$. BPF implementations that resample using this expensive $O(mn)$ algorithm can afford to resample only sporadically; this represents a difficult engineering tradeoff between BPF quality and computational cost.

In this paper we first present an algorithm with complexity $O(m + n \log m)$ based on a binary heap-structured tree which is straightforward, has good performance, and requires no special math. This is essentially a binary search method, often used in faster BPF implementations [**?**]. However, we introduce a potential performance optimization that takes advantage of the heap property.

Resampling via binary search is often used in faster BPF implementations [**?**]. However, since the other steps in a BPF iteration are linear in the number of particles, a $O(m+n\log m)$ binary search will still be the bottleneck step in such an implementation—a factor of 10 slowdown for a 1000-particle filter, representing significant extra computation. The memory access patterns of a binary search also interact badly with the cache of a modern processor.

We introduce a family of perfect and approximate algorithms based on the idea of simulating the scans of the naïve algorithm in parallel; making a single pass through the m input samples and selecting n output samples during this pass.

One well-known method that has been used to regain the performance lost to resampling is to give up on statistical correctness and simply sample at regular intervals [**?**]. We derive a standard approximate $O(m+n)$ algorithms for regular resampling. In practice, this seems to work well, and to run quite quickly. However, one cannot help but be a bit concerned that regular resampling will "go wrong" in a crucial situation, due to correlations between the sampling interval and an unfortunate particle set. Although the idea does not seem to be widely known, the resistance of regular resampling to such correlations can be improved simply by shuffling the particles before resampling. Since shuffling can be done in $O(n)$ time with good constant factors, this is probably a good idea.

The principal contribution of this work is an algorithm that we believe to be novel, with time complexity $O(m+n)$ (requiring just the normal $O(m+n)$ space) that produces a resampling statistically equivalent to naïve resampling. For a simple array representation of the output, simply initializing the output will require time $O(n)$. It seems that the input must be scanned at least once just to determine the total input weight for normalization. Thus, the running time of our algorithms is apparently optimal.

An $O(n)$ algorithm by Beadle and Djuric [**?**] produces an output sample that is only approximately correctly weighted. The basic method is to repeatedly randomly select an input sample, and then replicate it in the output sample the number of times that it would be expected to appear based on its fraction of the total weight. As noted above, calculating the total weight requires $O(m)$ time, so the advantage of the $O(n)$ algorithm over an $O(m+n)$ one is essentially lost in practice. The quality of the approximation is shown to be reasonably good, and it is shown to work well in one BPF application. However, it is only an approximate method, and one might expect that there are limits to the approximation quality. The authors report a speedup of approximately 20 for 500 particles, but no other details are given about running time. We suspect that the situation is more complicated; to investigate further would require independently implementing and timing this approach.

BPF is typically computation-limited, and all of the other steps in a BPF iteration require time linear in the population size. By linearizing resampling, we remove the resampling bottleneck, allowing higher population sizes that in turn dramatically improve BPF performance.

In the remainder of this paper we review existing algorithms, describe our algorithms, and report the effectiveness of algorithms in a BPF implementation. We conclude with a discussion of various issues.

```
to sample(μ):
    t ← 0
    for i from 1 to m do
            t ← t + w(s_i)
            if t > μ then
                    return s_i

to resample:
    for i from 1 to n do
            μ ← random-real([0..1])
            s'_i ← sample(μ)
```

Figure 1: Naïve Resampling

2 Weighted Resampling Algorithms

There are a number of approaches to the weighted resampling problem. In this section, we describe some weighted resampling algorithms in order of increasing time efficiency. We conclude with the description of some $O(m+n)$ algorithms.

For what follows, assume an array s of m input samples, and an output array s' that will hold the n output samples of the resampling. Assume further that associated with each sample s_i is a weight $w(s_i)$, and that the weights have been normalized to sum to 1. This can of course be done in time $O(m)$, but typical efficient implementations keep a running weight total during weight generation, and then normalize their sampling range rather than normalizing the weights themselves. We thus discount the normalization cost in our analysis.

2.1 A Naïve $O(mn)$ Resampling Algorithm

The naïve approach to resampling has been re-invented many times. A correct, if inefficient, way to resample is via the pseudocode of Figure 1. The *sample* procedure selects the first sample such that the sum of weights in the input up to and including this sample is greater than some index value μ. The index value is chosen in *resample* by uniform random sampling from the distribution [0..1], with each output position being filled in turn.

The naïve algorithm has its advantages. It is easy to verify that it is a perfect sampling algorithm. It is easy to implement, and easy to parallelize. The expected running time is $o(\frac{1}{2}mn)$. If the distribution of weights is uneven enough, as is typical with BPF, the proportionality constant can be improved by paying $O(m \log m)$ time up front to sort the input array so that the largest weights occur first. Regardless, naïve resampling is still the bottleneck step in BPF implementations that employ it.

```
to init-weights:
    for i from m downto 1 do
        l ← 2i
        r ← 2i + 1
        if l > m then
            w_t(s_i) ← w(s_i)
        else if r > m then
            w_t(s_i) ← w_t(s_l) + w(s_i)
        else
            w_t(s_i) ← w_t(s_l) + w(s_i) + w_t(s_r)
```

Figure 2: Computing Weights of Sub-heaps

```
to sample(μ, i):
    l ← 2i
    r ← 2i + 1
    if μ < w_t(s_l) then
        return sample(μ, l)
    if μ ≤ w_t(s_l) + w(s_i) then
        return s_i
    return sample(μ − w_t(s_l) − w(s_i), r)
```

Figure 3: Heap-based Sampling

2.2 A Heap-based $O(m + n \log m)$ Resampling Algorithm

One simple way to improve the performance of the naïve algorithm is to improve upon the linear scan performed by *sample* in Figure 1.

One way to do this is to treat the input sample array as a binary heap. In time $O(m)$ we can compute and cache the sum w_l of weights of the subtree at each position in the input, as shown in Figure 2. The sum at each heap position is computed bottom-up and stored as w_t.

Given w_t, *sample* can perform a scan for the correct input weight in time $O(\log m)$ by scanning down from the top of the heap, as shown in Figure 3. At each step, if the target variate μ is less than the weight of the left subtree, the scan descends left. If μ is greater than the weight of the left subtree by less than the weight of the current node the scan terminates and this node is selected. Otherwise, the scan descends right, with μ adjusted downward by the cumulate weight.

This algorithm is a bit more complex than the naïve one, but it dramatically improves upon the worst-case running time.

As with the naïve algorithm, a small constant-factor improvement is possible by actually heapifying the input such that the largest-weighted inputs are near the top. Heapification is also $O(m)$ time and can be done in place, so there is no computational complexity penalty for this

```
to merge(u):
    j ← 1
    t ← u_1
    for i from 1 to n do
        μ ← u_i
        while μ < t do
            t ← t + w(s_j)
            j ← j + 1
        s'_i ← s_j
```

Figure 4: Merge-based Resampling

optimization. However, the constant factors must be carefully balanced; our experiments show a small net loss in some situations and net gain in others. The controlling factor here is the distribution of weights; if a few samples carry most of the sample weight, heapification will pay for itself. Since the case in BPF where a few samples carry most of the weight tends to be the case where the accuracy of the filter is low, this optimization may be of special benefit in a variable-population BPF technique such as KLD-sampling [**?**].

For the normal case of resampling, we would like to get rid of the $\log m$ penalty per output sample. However, there is a rarely-occuring special case in which this algorithm is especially efficient. Consider an offline sampling problem in which we plan to repeatedly draw a small number of samples from the same extremely large input distribution. Because the input distribution remains fixed, the cost of heapification can be amortized away, yielding an amortized $O(n \log m)$ algorithm.

2.3 A Merge-based $O(m + n \log n)$ Resampling Algorithm

One can imagine trying to improve upon the complexity of the heap-style algorithm by using some more efficient data structure. However, there is a fundamental tradeoff—the setup for an improved algorithm needs to continue to have a cost low in m. Otherwise, any savings in resampling will be swamped by setup in the common case that $m \approx n$.

A better plan is to try to improve the naïve algorithm in a different way. The real problem with the naïve algorithm is not so much the cost per scan of the input as it is the fact that each scan is independent. It seems a shame not to try to do all the work in one scan.

Let us generate an array u of n variates up-front, then sort it. At this point, a *merge* operation, as shown in Figure 4, can be used to generate all n outputs in a single pass over the m inputs. The merge operation is simple. Walk the input array once. Each time the sum of weights hits the current variate u_i, output a sample and move to the next variate u_{i+1}. The time complexity of the initial sort is $O(n \log n)$ and of the merge pass is $O(m + n)$, for a total time complexity of $O(m + n \log n)$.

Complexity-wise, we seem to have simply moved the log factor of the previous algorithm from m to n, replacing our $O(m+n\log m)$ algorithm with an $O(m+n\log n)$ one. However, our new algorithm has an important distinction. The log factor this time comes merely from sorting an array of uniform variates. If we could somehow generate the variates in sorted order (at amortized constant cost) we could make this approach run in time $O(m+n)$. The next section shows how to achieve this.

Alternatively, if we could sort the variates in $O(n)$ time, we could get an $O(m+n)$ merge-based algorithm. Given that what we are sorting is n variates uniformly distributed between 0 and 1, a radix sort should do the trick here. However, the constant factors in the running time are expected to be poor for large n, since the radix sort has extremely poor cache behavior.

2.4 Regular Approximate Resampling

For large m, a sorted array of uniform deviates looks an awful lot like it was produced by sampling at regular intervals; the spacing between deviates is pretty uniform. This suggests an obvious approximate linear-time algorithm, which turns out to be a well-known approach [**?**]: sample the n output samples at uniformly-spaced regular intervals. While it is not clear that such a resampling algorithm has the same statistical properties as a perfect resampling, it seems to be sufficiently good for our BPF implementation, and to run about as fast as possible.

If one is concerned about the possible selection errors of regular sampling due to correlation, one might choose to shuffle the sample array prior to sampling. Since the shuffle can be performed in $O(m)$ time, there is no asymptotic penalty.

2.5 An Optimal $O(m+n)$ Resampling Algorithm

As discussed in Section 2.3, if we could generate the variates comprising a uniform sample of n values in increasing order, we could resample perfectly in time $O(m+n)$, improving upon the regular method of Section 2.4. In this section, we show an approach that achieves this goal.

Assume without loss of generality that our goal is simply to generate the first variate in a uniform sample of $n+1$ values. Call the first variate μ_0, the set of remaining variates U and note that $|U| = n$. Now, for any given variate $\mu_i \in X$, we have that

$$\Pr(\mu_0 < \mu_i) = 1 - \mu_0$$

Since this is independently true for each μ_i, we define

$$p(\mu_0) = \Pr(\forall \mu_i \in U \;.\; \mu_0 < \mu_i) = (1-\mu_0)^n$$

To see how to take a weighted random sample from this distribution, first consider a discrete approximation that divides the range 0..1 into T intervals. Given a random variate μ, we will

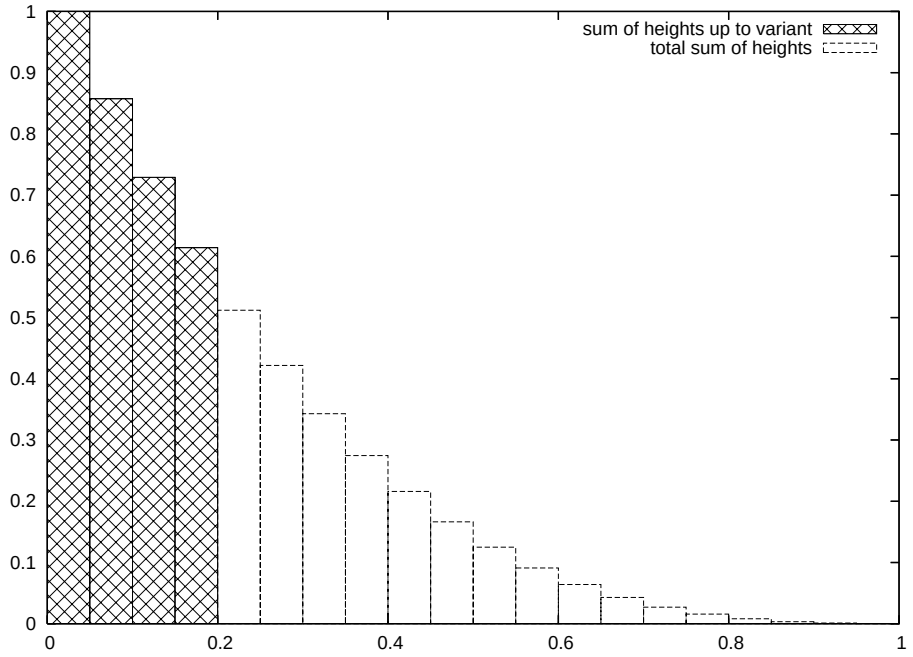

Figure 5: Discrete Sampling Approximation

use $\mu_0 = i_0/T$ when the sum of the weights of the intervals up to interval i_0 is just greater than μ. Of course, the weights must be normalized by dividing by the total weight. Thus we have

$$i_0 = \min_{i \in 1..T} \left[\frac{\sum_{j=1}^{i} p(j/T)}{\sum_{j=1}^{T} p(j/T)} \geq \mu \right]$$

Figure 5 illustrates the calculation here. We output as our weighted variate the x-coordinate i_0/T of the bar containing μ.

In the limit as $T \to \infty$ our discrete approximation converges to an integral. We have

$$\begin{aligned}
\mu &= \frac{\int_{u=0}^{\mu_0} (1-u)^n \mathrm{d}u}{\int_{u=0}^{1} (1-u)^n \mathrm{d}u} \\
&= \frac{\left. \frac{-(1-u)^{n+1}}{n+1} \right|_{u=0}^{\mu_0}}{\left. \frac{-(1-u)^{n+1}}{n+1} \right|_{u=0}^{1}} \\
&= \frac{\frac{-1}{n+1}\left[(1-\mu_0)^{n+1} - 1\right]}{0 - \frac{-1}{n+1}} \\
&= 1 - (1-\mu_0)^{n+1}
\end{aligned}$$

to *randomize*:
$u_1 \leftarrow (1-\mu)^{\frac{1}{n}}$
for i **from** 2 **to** n **do**
$u_i \leftarrow u_{i-1} + (1-u_{i-1})(1-\mu)^{\frac{1}{n-i+1}}$

Figure 6: Generating Deviates In Increasing Order

However, what we need is μ_0 in terms of μ, so we solve

$$\begin{aligned} \mu &= 1-(1-\mu_0)^{n+1} \\ (1-\mu_0)^{n+1} &= 1-\mu \\ \mu_0 &= 1-(1-\mu)^{\frac{1}{n+1}} \\ \mu_0 &= 1-\mu^{\frac{1}{n+1}} \end{aligned}$$

(The last step is permissible because μ is a uniform deviate in the range 0..1, and therefore statistically equivalent to $(1-\mu)$.)

We now have the formula we need for selecting the first deviate from a set of n in increasing order. To select the next deviate, we simply decrease n by 1, select a deviate from the whole range, and then scale and offset it to the remaining range. We repeat this process until $n = 0$. (Recall that $|U| = n$, so the last deviate will be selected when $n = 0$.) Figure 6 shows this process.

We now have the array u of deviates in sorted order that we need to feed the *merge* algorithm of the previous section. We thus have an $O(m+n)$ algorithm for random weighted selection.

2.5.1 Correctness of the Optimal Algorithm

To see that our optimal algorithm is correct, we first must describe what correctness means. To do this, we will define *statistical equivalence* of distributions, and then show that the distribution produced by the optimal algorithm is statistically equivalent to that produced by the naïve algorithm.

Definition 1 *A* distribution *is a tuple* $\langle S, p\rangle$ *where* S *is a nonempty, possibly infinite set, and* $p : S \to \mathcal{R}$ *is a probability function on elements of the set normalized such that*

$$\sum_{s\in S} p(s) = 1$$

Definition 2 *Two distributions* $\langle S, p_1\rangle$ *and* $\langle S, p_2\rangle$ *over the same set* S *are* statistically equivalent *when the probability of each element is equal,* i.e., *when*

$$\forall s \in S \;.\; p_1(s) = p_2(s)$$