FAST PERFECT WEIGHTED RESAMPLING

Bart Massey

Associate Professor
Computer Science Department
Portland State University
Portland, Oregon USA
bart@cs.pdx.edu

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.

Index Terms— Monte Carlo methods, state estimation, filtering, tracking filters

1. INTRODUCTION

Bayesian Particle Filtering (BPF) [1] 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.

Resampling via binary search is often used in faster BPF implementations [2]. 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.

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 [3]. 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.

What is really wanted, however, is a resampling algorithm with the $O(m + n)$ running time of regular resampling, but statistically equivalent to naïve resampling. We give such an algorithm, which we believe to be novel. 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. We also report on implementations that are performance-competitive with regular resampling in a toy BPF domain.

2. WEIGHTED RESAMPLING 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 ran-

```
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)
```

Fig. 1. Naïve Resampling

dom sampling from the distribution $[0..1]$, with each output position being filled in turn.

Despite its poor performance, 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 $o(\frac{1}{2}mn)$.

To derive a $O(m + n \log m)$ algorithm from the naïve algorithm, note that the linear scan of input samples in Figure 1 can be replaced with a binary search. One way to do this would be to treat the array of input samples as a heap. This heap-based algorithm, not shown here for space reasons, does dramatically improve on the performance of the naïve algorithm without sacrificing correctness.

For some input particle distributions, a further constant-factor improvement to both the naïve and heap-based algorithms can be had by sorting or heapifying, respectively, the input particle array so that the largest particles are likely to be encountered first in the search. The amortized cost of these operations is small, but may be larger than the cost savings in typical distributions; the approach also adds a bit to the complexity of the implementation.

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

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 2, 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 heap-based algorithm from m to n, replacing an $O(m + n \log m)$ algorithm with an $O(m + n \log n)$ one.

```
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
```

Fig. 2. Merge-based Resampling

However, the 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.

2.3. An Optimal $O(m + n)$ Resampling Algorithm

As discussed in the previous section, if we can generate the variates comprising a uniform sample of n values in increasing order, we can resample in time $O(m + n)$. 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

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

Thus, if we successively generate n variates u_i drawn from the distribution $(1-\mu_0)^{n-i}$, those variates will be statistically indistinguishable from the set of variates produced by generating n uniform variates and then sorting them. To generate a variate from the target distribution, it is sufficient to observe that the likelihood of generating a variate μ is given by

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

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

Fig. 3. 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 3 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.

2.4. Faster Generation of Variates

The method of directly generating variates described in the

previous section has a minor limitation: it requires an exponentiation per variate. Even a fast implementation of the mathematical function `pow()` on a machine with fast floating point is expensive compared to the single multiply of regular resampling.

One possible way around the `pow()` bottleneck is to generate random variates with distribution $(1-x)^n$ by Marsaglia and Tsang's "Ziggurat Method" PRNG [4, 5]. Unfortunately, the desired distribution is bivariate. The most direct approach would lead to generating n sets of Ziggurat tables, which would be prohibitively memory-expensive for large n.

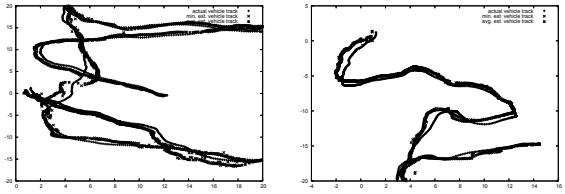
When computing $(1-x)^n$ for large n, however, our probability expression becomes self-similar, and we can accurately approximate the function for larger n using the function with smaller n. In fact, this is the well-known compound interest problem, yielding an elegant limiting approximation.

$$\lim_{a\to\infty}\left(1-\frac{x}{a}\right)^{an} = \lim_{a\to\infty}\left(1+\frac{(-x)}{a}\right)^{an} = e^{-xn}$$

This approximation corresponds to a standard linear-time approximate resampling method in which the next sample weight is given by an exponentially-distributed variate [6]. The approximation works well up until near the end of the resampling, and is relatively inexpensive if a Ziggurat-style exponential generator is used.

We can get the same performance with perfect resampling, though, by modifying a Ziggurat generator for $e^{-50\mu_0}$ to accurately compute the desired power by tweaking the rejection step. The efficiency of the generator would deteriorate unacceptably in the last 50 samples, so at that point we just switch to calling `pow()` directly. The resulting resampling implementation has performance close to that of regular resampling, but with the perfect resampling of the naïve method.

A parallel version of our perfect algorithm is straightforward to achieve. Although space precludes a detailed description, the basic idea is as follows. Given p processors, each processor generates a variate v_i from the distribution $x^i(1-x)^{p-i}$, then binary searches for the sample breakpoint corresponding to that variate. Then the processor generates n/p samples in the range $v_{i-1} \ldots v_i$ using the perfect resampling algorithm. This achieves a linear speedup while retaining statistically correct resampling.



(a) Naïve Resampling (b) Optimal Resampling

Fig. 4. Vehicle Tracking Using BPF

3. EVALUATION

We implemented the algorithms described previously in a BPF tracker for simulated vehicle navigation. The simulated vehicle has a GPS-like and an IMU-like device for navigational purposes; both the device and the vehicle model are noisy. Figures 3 and 3 show 1000-step actual and estimated vehicle tracks from the simulator for the 100-particle case, using the naïve and optimal algorithms. Note that the quality of tracking of the two algorithms is indistinguishable, as expected.

The important distinction between the algorithms we have presented is not quality, but rather runtime. Figure 5 shows the time in seconds for 1000 iterations of BPF with various resampling algorithms as a function of the number of particles tracked / resampled. The benchmark machine is an otherwise unloaded Intel Core II Duo box at 2.13 GHz with 2GB of memory.

As expected, BPF using the naïve algorithm becomes unusable at larger particle sizes, whereas BPF using the optimal

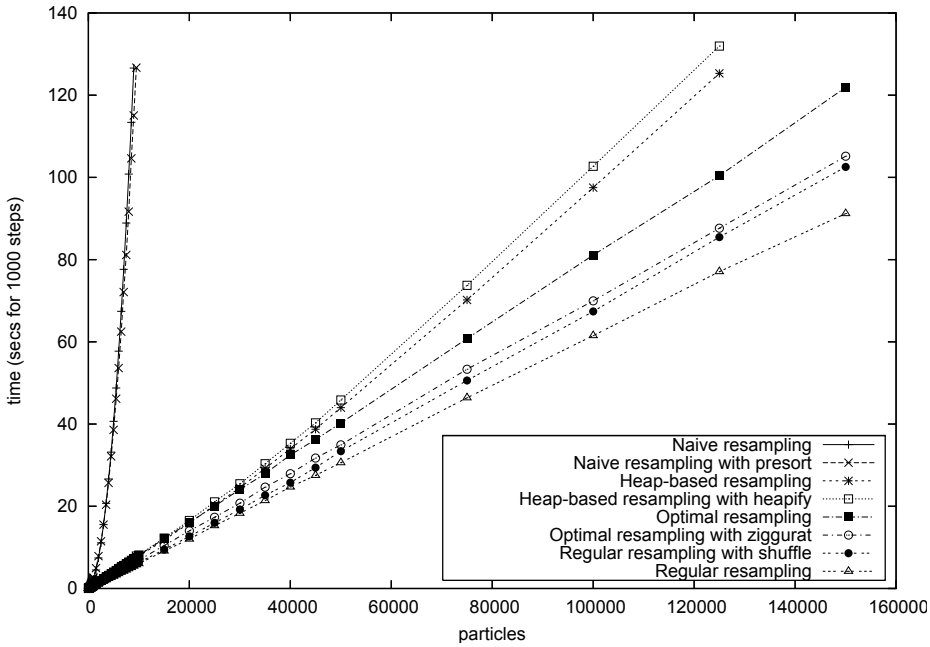


Fig. 5. Runtimes for BPF Resampling Implementations

algorithm scales linearly. The heap-based algorithms are surprisingly competitive with the optimal algorithm even at large particle counts, although the distinction is somewhat masked by the mediocre running time of the rest of the BPF implementation; resampling is not the bottleneck for any of these fast algorithms, as the near-zero cost of the regular algorithm without shuffling indicates.

The performance difference between our Ziggurat-based implementation of optimal resampling and our shuffled implementation of regular resampling is quite small. Given this, we believe that the statistical correctness of optimal resampling will make it the best choice for many implementations.

4. AVAILABILITY

Our C implementation of BPF with linear resampling described here is freely available under the GPL at `http://wiki.cs.pdx.edu/bartforge/bmpf`. It relies on our BSD-licensed implementation (partly based on the work of others—please see the distribution for attribution) of various PRNGs and Ziggurat generators at `http://wiki.cs.pdx.edu/bartforge/ziggurat`.

5. ACKNOWLEDGMENTS

Thanks much to Jules Kongslie, Mark Jones, Dave Archer, Jamey Sharp, Josh Triplett and Bryant York for illuminating conversations during the discussion of this work. Thanks also to Jules Kongslie and to James McNames and his students for patiently explaining BPF to me and answering my questions. Finally, thanks to Keith Packard and Intel for providing the hardware on which this work was primarily done.

6. REFERENCES

[1] Branko Ristic, Sanjeev Arulampalam, and Neil Gordon, *Beyond the Kalman Filter: Particle Filters for Tracking Applications*, Artech House, Feb. 2004.

[2] S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for on-line non-linear/non-Gaussian Bayesian tracking," *IEEE Transactions on Signal Processing*, vol. 50, no. 2, Feb. 2002.

[3] G. Kitagawa, "Monte Carlo filter and smoother for non-Gaussian nonlinear state space models," *Journal of Computational and Graphical Statistics*, vol. 5, no. 1, 1996.

[4] George Marsaglia and Wai Wan Tsang, "The ziggurat method for generating random variables," *J. Statistical Software*, vol. 5, no. 8, Oct. 2000.

[5] Boaz Nadler, "Design flaws in the implementation of the Ziggurat and Monty Python methods (and some remarks on Matlab randn)," 2006, arXiv.org.

[6] J. Carpenter, P. Clifford, and P. Fernhead, "An improved particle filter for non-linear problems," 1997.