Linear Time Weighted Resampling

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

Draft of 25 August 2007
Do Not Distribute

Abstract

We describe an optimal algorithm for perfect weighted resampling of a population, with time complexity $O(m + n)$ for resampling m inputs to produce n outputs. We also describe a perfect algorithm with time complexity $O(n + n \log m)$ which is highly suitable for fixed-point microcontroller implementations, and an extremely fast approximate $O(n + m)$ algorithm whose performance These algorithms represent a substantial performance improvement over the typically-used resampling algorithm, and a quality improvement over a linear-time approximate algorithm that represents the state of the art. Our resampling algorithm is also parallelizable, with linear speedup. Linear-time resampling yields substantial improvements in our motivating example of Bayesian Particle Filtering.

1 Introduction

Bayesian Particle Filtering (BPF) [3] is an exciting new methodology for state space tracking and sensor fusion. The bottleneck step in BPF is weighted 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. A standard naïve algorithm has time complexity $O(mn)$. Typical BPF implementations will resample using this expensive $O(mn)$ algorithm, but only sporadically; this represents a difficult engineering tradeoff between BPF quality and computational cost.

An $O(n)$ algorithm by Beadle and Djuric [1] 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. Note that 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, since it is only an approximate method, the results remain suspect. 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, and will probably independently implement and time this approach.

In this paper we first present an algorithm with complexity $O(m + n \log m)$ based on a binary heap which is straightforward, has good performance, and requires no special math, but seems not to be known in practice. We then 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 output samples using n successive uniform variates.

We introduce a novel algorithm with time complexity $O(m + n)$ (requiring just the normal $O(m+n)$ space) that produces a perfectly weighted sample. We modify our exact algorithm to produce an $O(m + n)$ time approximate algorithm with an extremely simple implementation that appears to work well. 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.

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 much higher population sizes that in turn dramatically improve BPF performance.

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

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 our $O(m + n)$ 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.

```
**to** $sample(\mu)$:
    $t \leftarrow 0$
    **for** $i$ **from** $1$ **to** $m$ **do**
        $t \leftarrow t + w(s_i)$
        **if** $t > \mu$ **then**
            **return** $s_i$

**to** $resample$:
    **for** $i$ **from** $1$ **to** $n$ **do**
        $\mu \leftarrow$ $random\text{-}real([0..1])$
        $s'_i \leftarrow sample(\mu)$
```

Figure 1: Naïve Resampling

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. Irregardless, naïve resampling is still the bottleneck step in a BPF implementation.

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.

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

Figure 2: Computing Weights of Sub-heaps

```
to sample($\mu, i$):
    $l \leftarrow 2i$
    $r \leftarrow 2i+1$
    if $\mu < w_t(s_l)$ then
        return sample($\mu, l$)
    if $\mu \leq w_t(s_l) + w(s_i)$ then
        return $s_i$
    return sample($\mu - w_t(s_l) - w(s_i), r$)
```

Figure 3: Heap-based Sampling

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 optimization.

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

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

Figure 4: Merge-based Resampling

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. We are currently experimenting with this approach; 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. In Section 4.1.2 we suggest a hybrid of the radix sort approach and a variant of the approach of the next section that is expected to largely solve that problem.

2.4 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 $O(m + n)$. In this section, we show the simple math that achieves this goal.

Assume without loss of generality that our goal is simply to generate the first variate in a

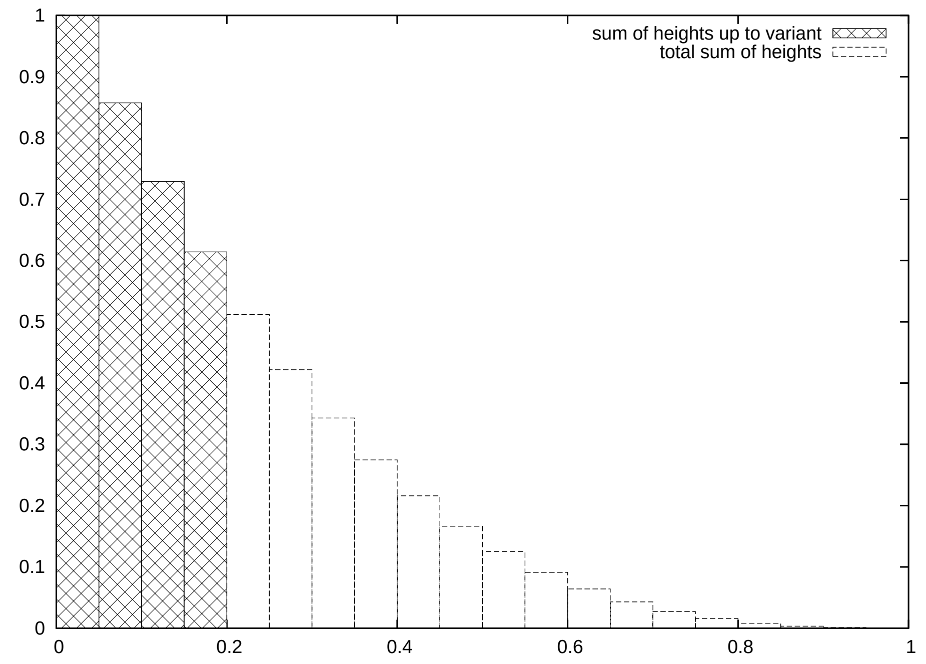

Figure 5: Discrete Sampling Approximation

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 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.

```
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

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

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

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 Regular Approximate Resampling

There are some obvious drawbacks to the optimal algorithm of the previous section. Most notably, the version presented there requires one call to a floating-point exponentiation function for each output sample. This can be quite expensive even on a fast machine; on a microcontroller it may not even be possible.

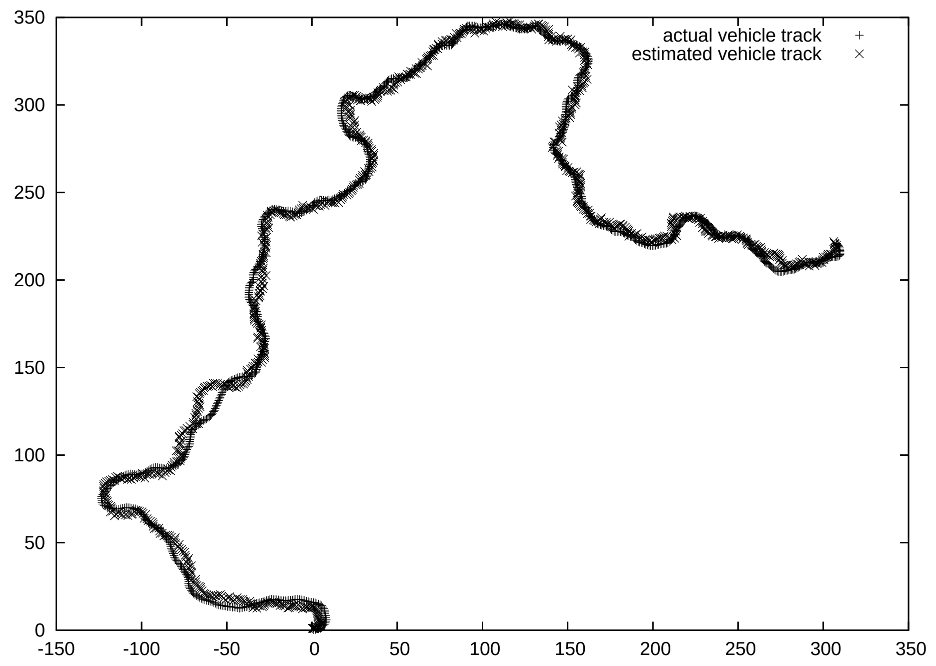

Figure 7: Vehicle Tracking—BPF With Naïve Resampling

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. Shuffle the input samples in time $O(m)$ to reduce bias, then 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.

3 Performance

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 7 and 8 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. Figures 9-11 show 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

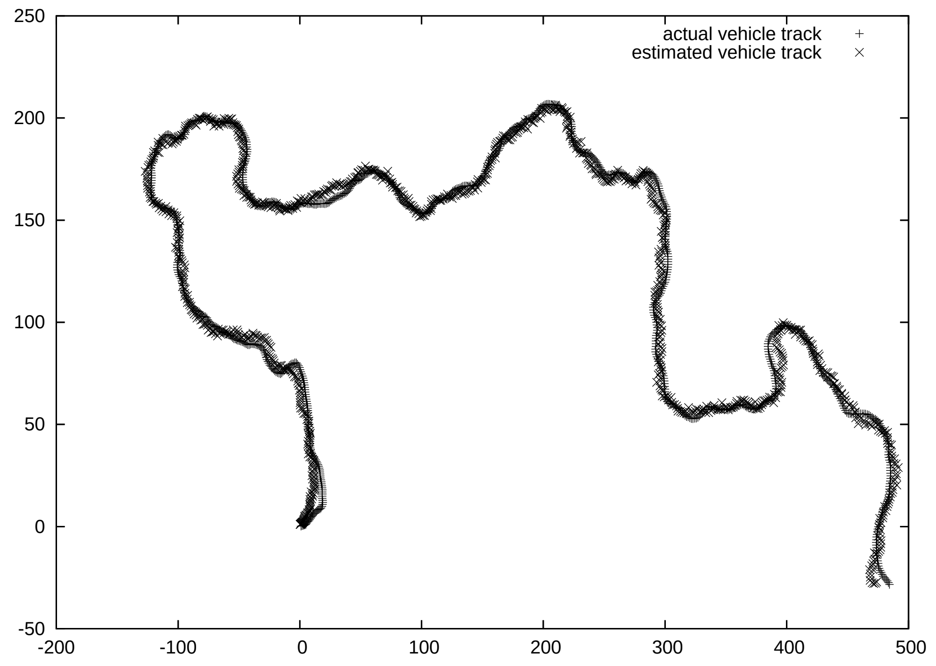

Figure 8: Vehicle Tracking—BPF With Optimal Resampling

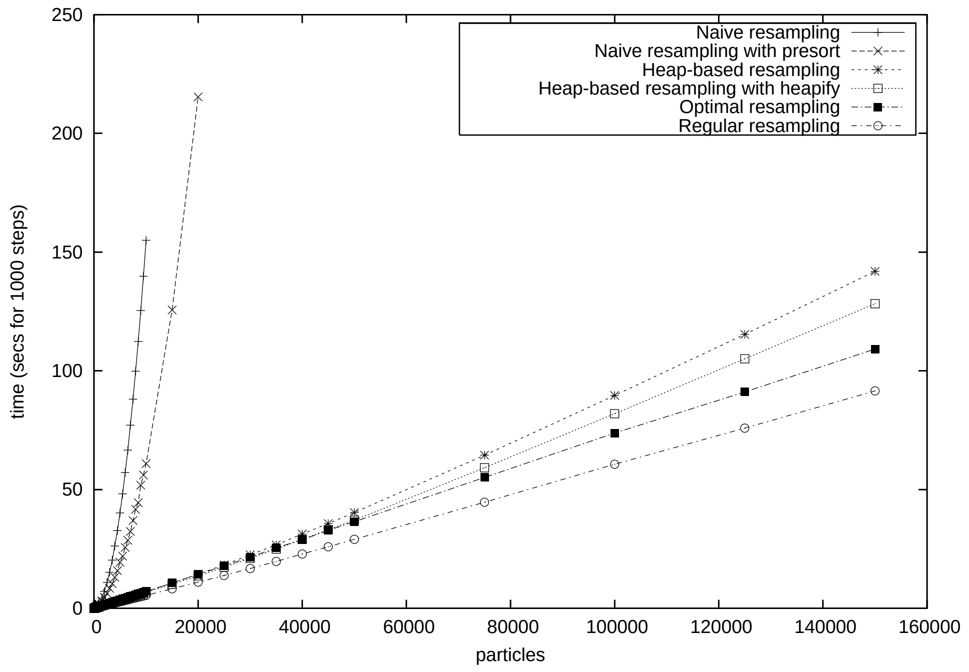

Figure 9: Runtimes for BPF Implementation

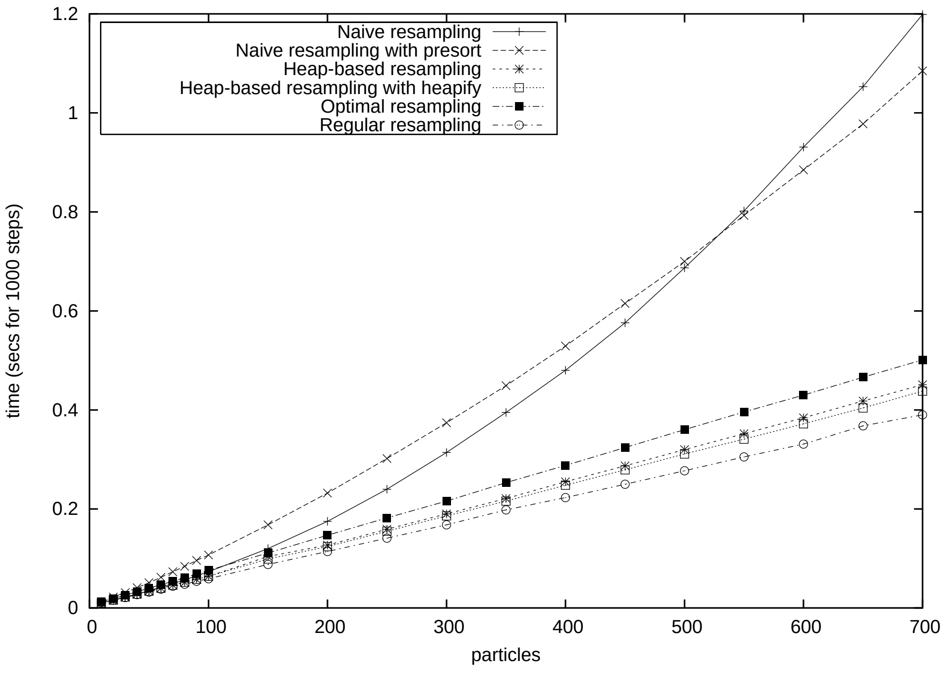

Figure 10: Detail of Figure 9

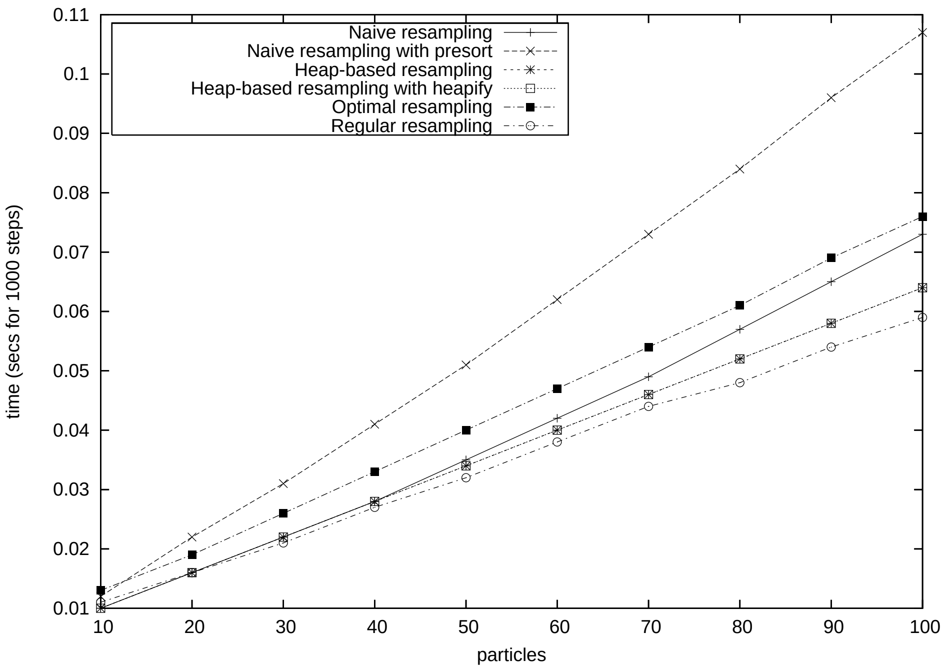

Figure 11: Detail of Figure 10

BPF using the optimal algorithm scales linearly. The heap-based algorithms are quite 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. The performance of the regular algorithm illustrates this—the cost of that algorithm is quite close to zero, and thus represents something of a bound on the performance improvement possible through faster resampling.

Sorting improves the performance of the naïve implementation substantially for larger n, although it does not pay until the particle count is around 500 (Figure 10). Heapification seems to always be worthwhile for the heap-based algorithm, though only significantly so for more than about 40,000 particles.

Our optimal algorithm is slightly out-performed by the heap-based algorithm until around 20,000 particles, and by the heap-based algorithm with heapification until around 40,000 particles. This, however, is reflective of the large constant factor we hope to eliminate in the optimal algorithm. Once this is done, the performance of optimal and regular resampling should be very similar.

4 Discussion

We have shown an $O(m + n)$ algorithm for perfect resampling, and an extremely fast BPF implementation based on this algorithm. However, further speedups are possible. In this section, we discuss some of them.

4.1 Constant Factors

In a typical noise model, there is one call to a Gaussian-distributed pseudo-random number generator and to the `exp()` function per particle *per sensor*: this is how the weights are updated. These are the constant-factor bottlenecks in a BPF implementation with linear-time or near-linear-time resampling.

The cost of generating Gaussian pseudo-random numbers can be reduced to insignificance by using Marsaglia and Tsang's "Ziggurat Method" PRNG [2]. We observed an approximate doubling of speed in our BPF implementation by switching to this PRNG, and it is now far from being the bottleneck.

The `exp()` bottleneck is harder. We have recently switched to a fast approximate exponentiation trick due to Schraudolph [4], which improved our performance considerably, but the large number of calls is still painful. It may be better to just change noise distributions altogether. Work is still underway in this area.

The remaining bottleneck in resampling itself is the cost of the $O(n)$ calls to the `pow()` function, each with a slightly different argument. These rational exponentiations are computed during variate generation by calling the standard math library function `pow(x,y)`. Since there is only one call to `pow()` per particle, but many calls to `exp()` per particle, this is not the bottleneck step for BPF. However, it would be nice to cut it down, and approaches are available.

4.1.1 Direct Generation of Variates

One way around the `pow()` bottleneck is to generate random variates with distribution $(1-x)^{n+1}$ by some less expensive method than that of Section 2.4. The Ziggurat method mentioned previously would be about right, except that the desired distribution is a two-argument function. The most direct approach would lead to generating n sets of Ziggurat tables, which would be prohibitively memory expensive for large n.

We suspect that there is a generic rejection method for variate generation that will work fine for large n. In this regime, our probability expression becomes self-similar, and we can very accurately approximate

$$(1-x)^{\frac{1}{an}} \approx \left(1 - \frac{x}{a}\right)^n$$

If this approximation is sufficiently accurate for our purpose, it could be used directly to eliminate most of the `pow()` calls by building a Ziggurat PRNG for $(1-x)^{n_0}$ for some reasonably large n_0—probably around 30 or more. The Ziggurat generator should by its nature replace almost all of the `pow()` calls with simple table multiplies and compares. This is the approach we are currently pursuing.

Alternatively, one could build error bounds for the approximation, and use these with a rejection method generator that also would very occasionally call `pow()` in a corner case.

4.1.2 Segmentation

The concentration so far has been on generating the uniform variates sequentially. However, there are at least two situations in which one would first like to break up the variates into blocks which are treated separately.

The first such situation is to combine direct generation with the radix sorting approach discussed at the end of Section 2.3. Imagine we could generate "breakpoints" between blocks of k variates for some small k, say 100, in constant time. We could then generate k variates uniformly and radix sort them in time $O(k)$. Because the blocks would be small, the radix sort should hit the L0 cache and be fast, yet we would divid the number of `pow()` calls by k over the direct approach of Section 2.4.

The second reason to break the variates up into blocks is for parallelization. This is discussed further in Section 4.2.

To break the variates up into blocks, we could try to derive a direct expression for placing sample k of n for arbitrary k. This would allow us to "skip forward" by k samples and place a variate, then deal with the intervening variates at our leisure.

We observe that if a uniform variate μ_k is at position k of n, $k-1$ of the samples must have landed to the left of μ_k and $n-k$ of the samples must have landed to the right. There are in general $\binom{n}{k-1}$ ways this can happen, so the probability of μ_k being the k^{th} sample is

$$p(\mu_k) = \Pr(\mu_k \text{ is at position } k) = \binom{n}{k-1}(\mu_k)^{k-1}(1-\mu_k)^{n-k}$$

The direct method used in Section 2.4 next requires computing the probability that a variate μ is in the right position via

$$p(\mu) = \Pr(\mu_k = \mu) = \frac{\int_{u=0}^{\mu_k} \binom{n}{k-1}(\mu_k)^k(1-\mu_k)^{n-k}\mathrm{d}u}{\int_{u=0}^{1} \binom{n}{k-1}(\mu_k)^k(1-\mu_k)^{n-k}\mathrm{d}u}$$

The integral in the denominator can be calculated directly.

$$\int_{u=0}^{1} \binom{n}{k-1}(\mu_k)^k(1-\mu_k)^{n-k}\mathrm{d}u = \binom{n}{k-1}\frac{\Gamma(1+n-k)\Gamma(k)}{\Gamma(n+1)}$$

Unfortunately, the integral in the numerator is more of a mess:

$$\int_{u=0}^{\mu_k} \binom{n}{k-1}(\mu_k)^k(1-\mu_k)^{n-k}\mathrm{d}u = \binom{n}{k-1}(\mu_k)^k\,{}_2F_1(k, -n+k; k+1; \mu_k)$$

where ${}_2F_1$ is Gauss's hypergeometric function.

Any further progress in this direction appears to be limited by the difficulty of solving $p(\mu)$ for μ_0. A rejection method for direction generation of μ_0 would probably be a better approach here—a somewhat inefficient method would be fine, since few calls would be made to this generator. Alternatively, one could just give up and divide the range uniformly. The error of this approximation should be extremely small, and it would avoid a lot of complexity.

4.2 Paralellization

Our optimal algorithm will parallelize reasonably well with some additional work.

1. Each processor fills in a section of total accumulated weights in the input particle array as described in Section 2.3.

2. In a separate pass, each processor adds the sum of weights computed by the processor to its left to its section of the total accumulated weights in the input particle array.

3. A master processor uses one of the methods of "skipping ahead" in the variate sequence described in the previous section to break up the array of variates to be calculated by our P-processor machine into P regions.

4. Each processor does a search for the start and end of its section of the input particle array—the section whose total weights contain its variates. This can be done in time $O(\log m)$ using binary search, with an impressively small constant factor.

5. Finally, each processor resamples its section of the array using any of the fast algorithms we describe. The key here is that these algorithms are all completely parallelizable with zero overhead, given the setup of steps 1–4.

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 teaching me BPF. It is my n^{th} attempt to learn the topic, but this time it worked.

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`.

References

[1] E.R. Beadle and P.M. Djuric. A fast weighted bayesian bootstrap filter for nonlinear model state estimation. *IEEE Trans. Aerospace and Electronic Systems*, 33(1), January 1997.

[2] George Marsaglia and Wai Wan Tsang. The ziggurat method for generating random variables. *J. Statistical Software*, 5(8), October 2000.

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

[4] Nicol N. Schraudolph. A fast, compact approximation of the exponential function. Technical Report IDSIA-07-98, Istituto Dalle Molle di Studi sull'Intelligenza Artificiale, Lugano, Switzerland, October 1998.