Linear Time Weighted Resampling

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

Draft of 24 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. This is a substantial performance improvement over typically-used resampling algorithms, and a quality improvement over the approximate linear-time algorithm that represent the state of the art. Our resampling algorithm is also easily 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 $O(mn)$. Algorithms with complexity $O(m+n \log m)$ are easy to find, although they seem not to be widely used in practice. Instead, typical BPF implementations will resample using the expensive $O(mn)$ algorithm, but only sporadically. A more promising approach is to use a linear time algorithm, but one that produces a sample that is only approximately correctly weighted [1].

We introduce an algorithm with time complexity $O(m+n)$ (requiring just the normal $O(m+n)$ space) that produces a perfectly weighted sample. 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, our algorithm is apparently optimal.

```
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

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.

A good starting point for describing our algorithm is to review some resampling algorithms. We describe our algorithm and report its effectiveness in a BPF implementation. We conclude with a discussion of remaining 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 $O(m+n)$ algorithm.

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.

```
to init-weights:
    for i from m downto 1 do
        if 2i > m then
            w_t(s_i) ← w(s_i)
        else if 2i + 1 > m then
            w_t(s_i) ← w_t(s_2i) + w(s_i)
        else
            w_t(s_i) ← w_t(s_2i) + w(s_i) + w_t(s_2i+1)
    for i from 1 to m do
        if 2i > m then
            w_l(s_i) ← 0
        else
            w_l(s_i) ← w_t(s_2i)
```

Figure 2: Computing Weights of Left Sub-heaps

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, the proportionality constant might be improved by paying $O(m \log m)$ time up front to sort the input array so that the largest weights occur first (but this does not seem to work well in practice). Irregardless, resampling is still the bottleneck step in a typical 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.

For example, imagine that the input array is treated as a binary heap. In time $O(m)$ we can compute and cache the sum w_l of weights of the left sub-heap of each position in the input, as shown in Figure 2. First, the sum at each heap position is computed bottom-up and stored as w_t. This then gives w_l as simply the w_t of the left child.

Given w_l, *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.

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, constant-factor improvements might be 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. However, it is an extra step, and the constant factors

```
to sample(μ, i):
    if μ ≤ w_l(s_i) then
        return sample(μ, 2i)
    if μ ≤ w_l(s_i) + w(s_i) then
        return s_i
    return sample(μ, 2i + 1)
```

Figure 3: Heap-based Sampling

may not be worth it.

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 quite 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 as it is the fact that each scan is independent. It seems a shame not to reuse the work of the initial scan in subsequent scans.

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

```
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

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 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.

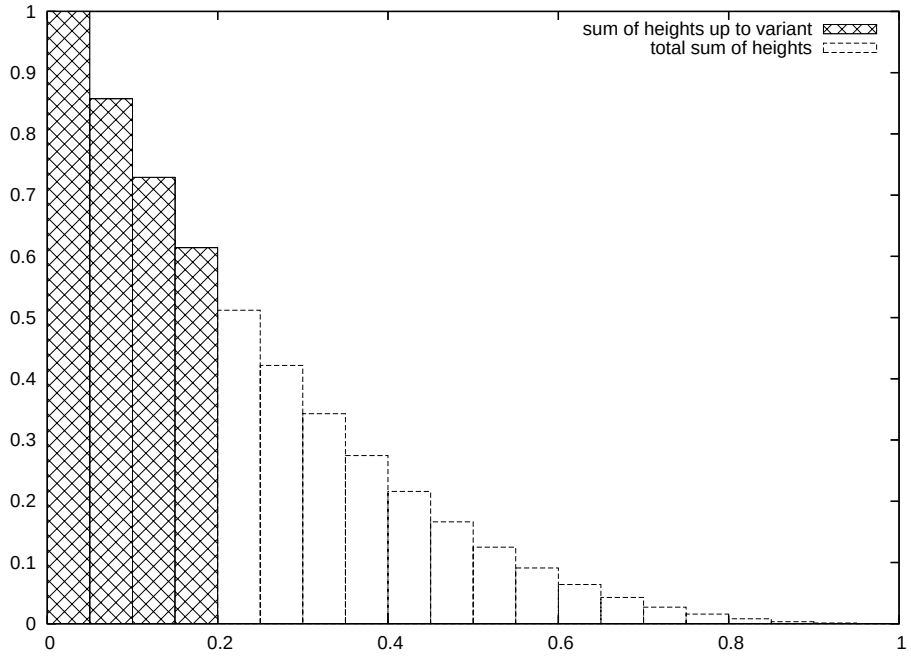

Figure 5: Discrete Sampling Approximation

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

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

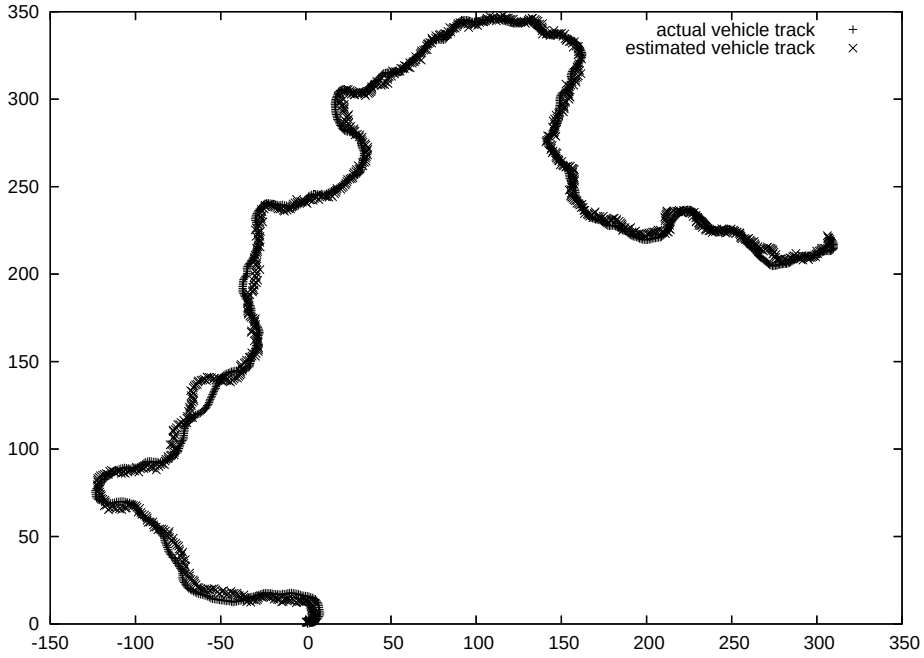

Figure 7: Vehicle Tracking—BPF With Naïve Resampling

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.

3 Performance

We implemented the naïve and optimal algorithms in a BPS 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 actual and estimated vehicle tracks from the simulator for the 100-particle case. Note that the quality of tracking of the two algorithms is indistinguishable, as expected.

The important distinction between these algorithms is not quality, but rather runtime. Figure 9

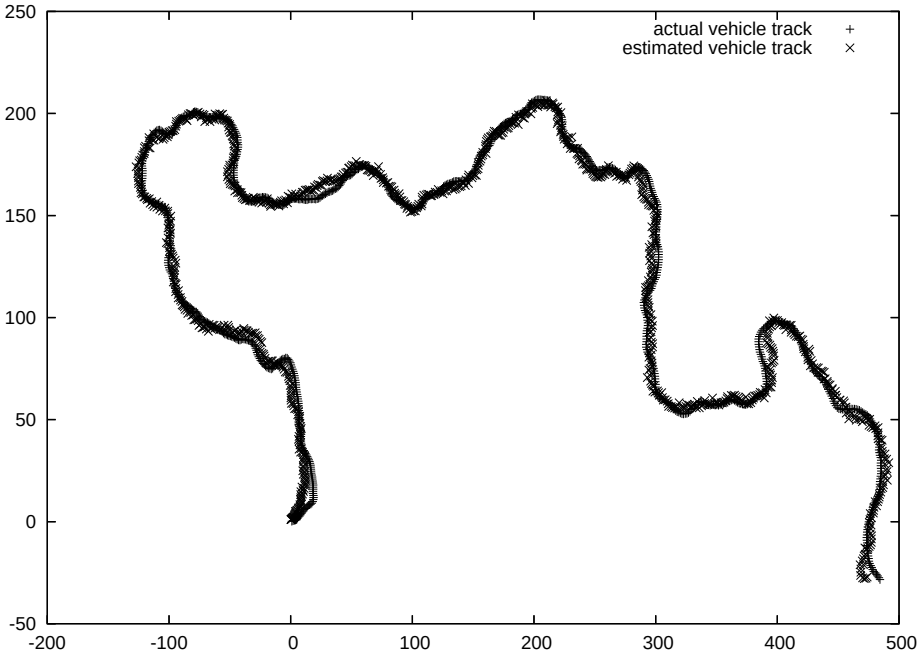

Figure 8: Vehicle Tracking—BPF With Optimal Resampling

shows the time in seconds for the naïve and optimal algorithms as a function of the particle size. The machine is an 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 algorithm scales linearly.

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. This is the current constant-factor bottleneck in the BPF implementation.

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.

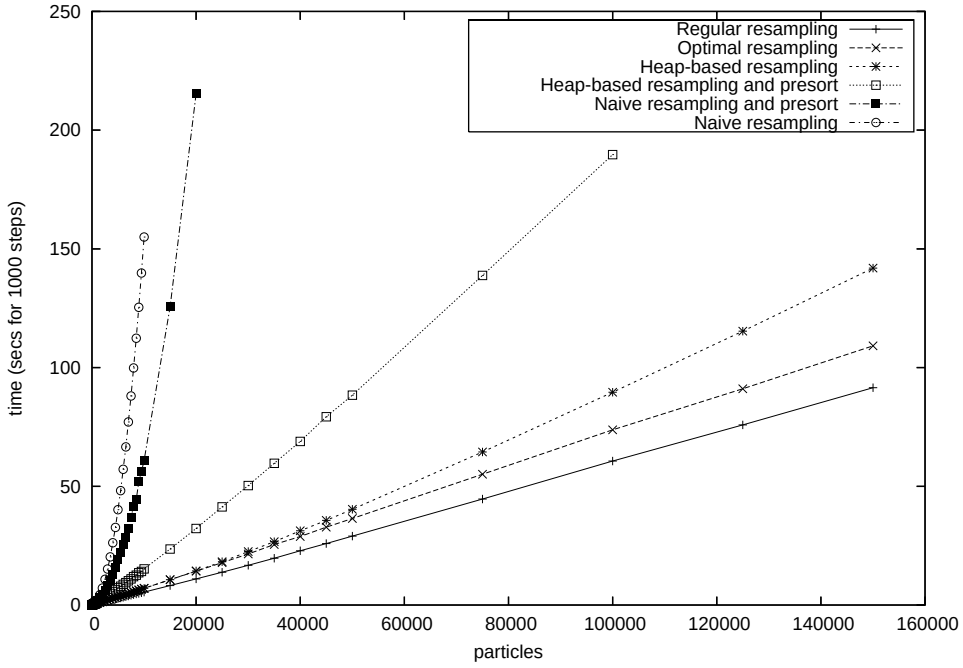

Figure 9: Runtimes for BPF Implementation

The `exp()` bottleneck is harder. One can always use approximate functions, or 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

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

If this approximation is sufficiently accurate for some 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.

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.

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 Section 4.1.2 to break up the array of variates to be calculated by our P-processor machine into P regions.
4. Each processor indepedently fills in its section of variates in the manner described in Section 2.4.
5. 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.
6. Finally, each processor samples its section of the array using the *merge* algorithm of Section 2.3.

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.