# Fast Perfect Weighted Resampling

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

Draft of December 22, 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) [8] 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 [1]. 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 [1]. 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 [5]. 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 [2] 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($\mu$):
    $t \leftarrow 0$
    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-real([0..1])
        $s'_i \leftarrow$ sample($\mu$)
```

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 $\mu$. 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
        $l \leftarrow 2i$
        $r \leftarrow 2i+1$
        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

## 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 $\mu$ is less than the weight of the left subtree, the scan descends left. If $\mu$ 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 $\mu$ 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 $merge(u)$:
    $j \leftarrow 1$
    $t \leftarrow u_1$
    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

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 [4].

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 [5]: 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 $\mu_0$, 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 $\mu_i$, 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 $\mu$, we will

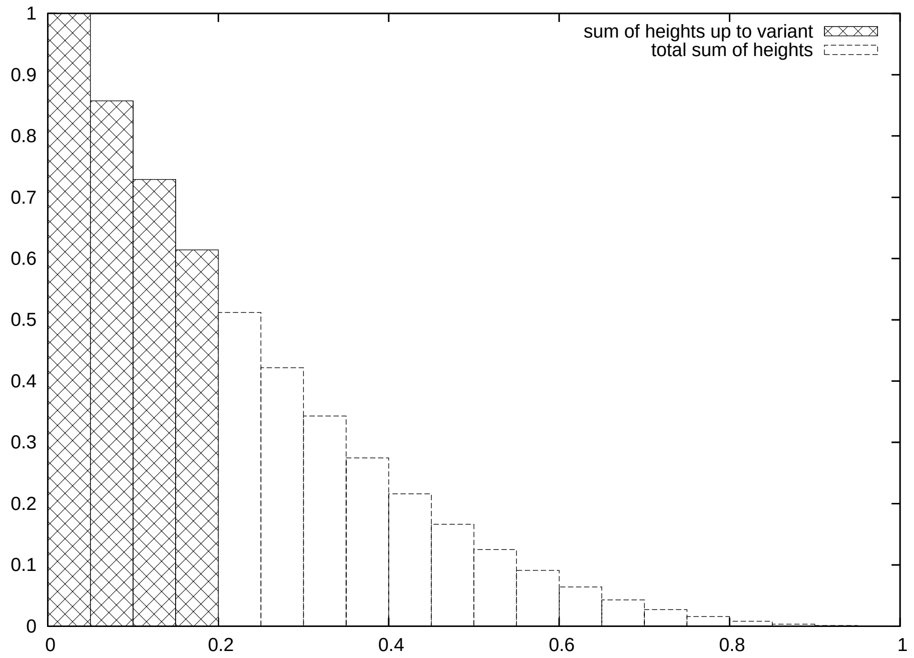


Figure 5: Discrete Sampling Approximation

use $\mu_0 = i_0/T$ when the sum of the weights of the intervals up to interval $i_0$ is just greater than $\mu$. 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 $\mu$.

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 $\mu_0$ in terms of $\mu$, 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 $\mu$ 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 uniform.

**Definition 1** *A* distribution *is a tuple $\langle S, p\rangle$ where $S$ is a nonempty, possibly infinite set, and $p : S \rightarrow \mathcal{R}$ is a probability density 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*

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

```
to randomize $n$ points:
    if $n = 0$ return $\langle\rangle$
    $u_0 \leftarrow (1-\mu)^{\frac{1}{n}}$
    $u = randomize(n-1)$
    return $\langle u_0\rangle \cdot (u_0 + (1-u_0)\cdot u)$
```

Figure 7: Recursively Generating Deviates

**Definition 3** *The set $U_n$ of n*-point uniform distributions *is defined by*

$$U_n(a,b) = \left\{\left\langle x, \frac{1}{b-a}\right\rangle | x \in [a..b]\right\}$$

We proceed by showing that $U_n(a,b)$ is linear in $a$ and $b$.

**Lemma 1 (Scaling)** *For all $s \in \mathcal{R}^+$, the distributions $U_n(0,s)$ and $s \cdot U_n(0,1)$ are statistically equivalent.*

**Lemma 2 (Offset)** *For all $t, a, b \in \mathcal{R}$, the distributions $U_n(t+a, t+b)$ and $t + U_n(a,b)$ are statistically equivalent.*

For convenience of proof, we note that Figure 6 is actually an iterative version of a more natural recursive algorithm show in Figure 7. The equivalence of the algorithms of Figures 6 and 7 is left as an exercise, but is straightforward.

The correctness of the recursive algorithm of Figure 7 can be established by a simple inductive proof. We sketch that proof here.

**Proposition 1** *The algorithm of Figure 7 generates a sorted uniform sequence of n points,* i.e., *an ordered vector drawn from the uniform distribution $U_n(0,1)$.*

**Proof Sketch:** *We proceed by induction on the number of points n being generated.*

***Base Case:*** *When $n = 0$, the sequence returned is a uniform sequence of 0 points. (This is a bit disingenuous. Note that the sequence returned when $n = 1$ is*

$$\langle (1-\mu)^{\frac{1}{1}}\rangle \cdot \langle\rangle = \langle 1-\mu\rangle = \langle\mu\rangle$$

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

***Inductive Case:*** *By construction, the variate $u_0$ is distributed in a fashion statistically equivalent to the leftmost variate drawn from a uniform n-point distribution. What remains to show is that the remaining $n-1$ variates are distributed properly.*

*To see this, consider what is known about an arbitrary point $u_j$. We know that $u_j$ was chosen uniformly over $0..1$, and that it turned out that $u_j \geq u_0$. A statistically equivalent process for generating $u_j$ would thus be to generate and reject uniform variates until we have one greater than or equal to $u_0$. Thus, $u_j$ can be drawn from the uniform distribution $U_{n-1}(u_0, 1)$. But by the scaling and offset lemmas above, this is statistically equivalent to drawing a $u$ from $U_{n-1}(0, 1)$ and then transforming it to $u_j = u_0 + (1 - u_0)u$. By the inductive hypothesis, our code draws $u$ from the uniform distribution; it then scales and offsets it in exactly the right way to distribute it across the remaining interval.*

### 2.5.2 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 [6, 7]. 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 $n$.

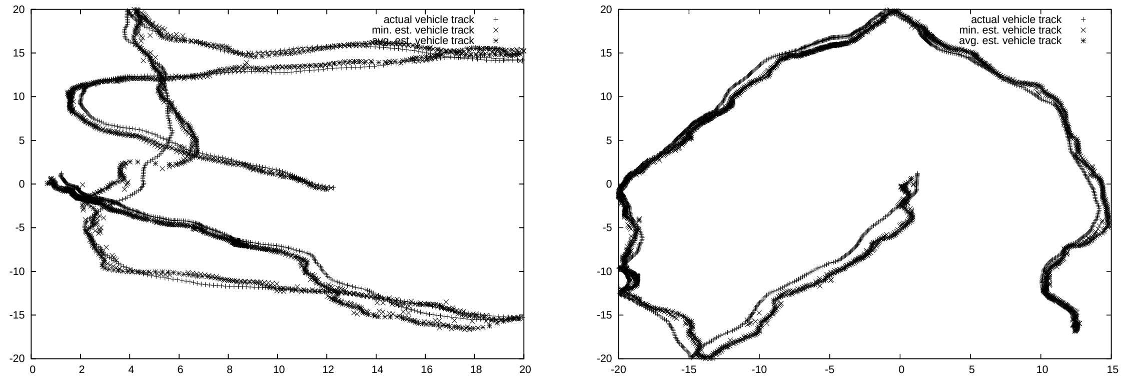
When computing $(1-x)^n$ for large $n$, however, our probability expression becomes self-similar, and we can accurately approximate the function for larger $n$ using the function with smaller $n$. In fact, this is the well-known compound interest problem, yielding an elegant limiting approximation.

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

This approximation corresponds to a standard linear-time approximate resampling method in which the next sample weight is given by an exponentially-distributed variate [3]. 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



(a) Naïve Resampling

(b) Optimal Resampling

Figure 8: Vehicle Tracking Using BPF

retaining statistically correct resampling.

# 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. Figures 9 and 10 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.

The distinction between algorithms 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 without shuffling 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.

As expected, BPF using the naïve algorithm becomes unusable at larger particle sizes, whereas BPF using the optimal algorithm scales linearly. There does not appear to be much advantage in using a heap-based algorithm at any particle size. The Ziggurat implementation of our optimal algorithm is quite close to regular resampling with shuffle at large particle counts. Given the correctness of our optimal algorithm, we think that this represents a significant achievement.

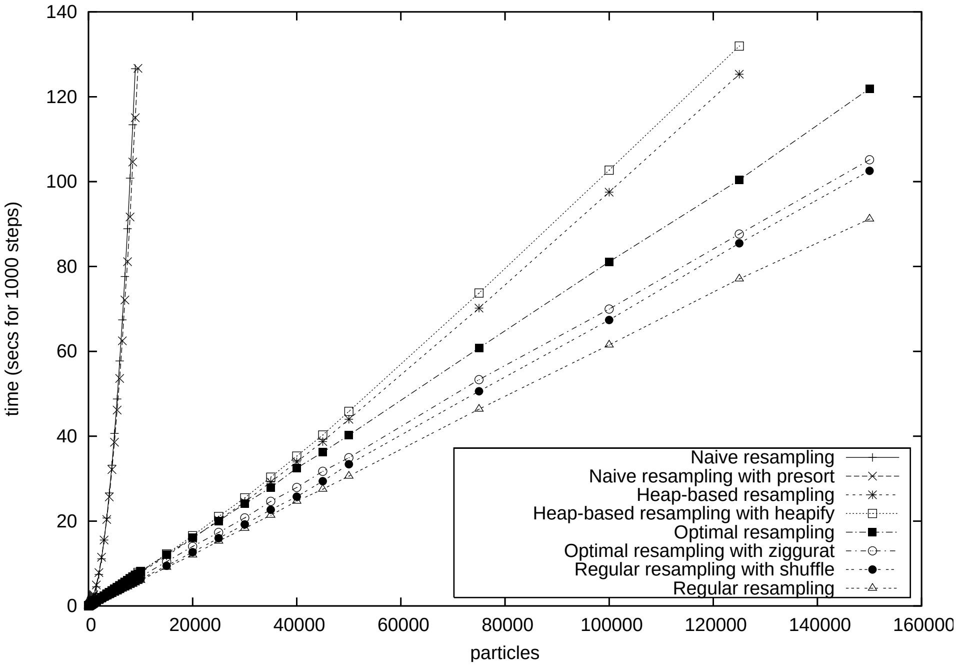


Figure 9: Runtimes for BPF Implementation

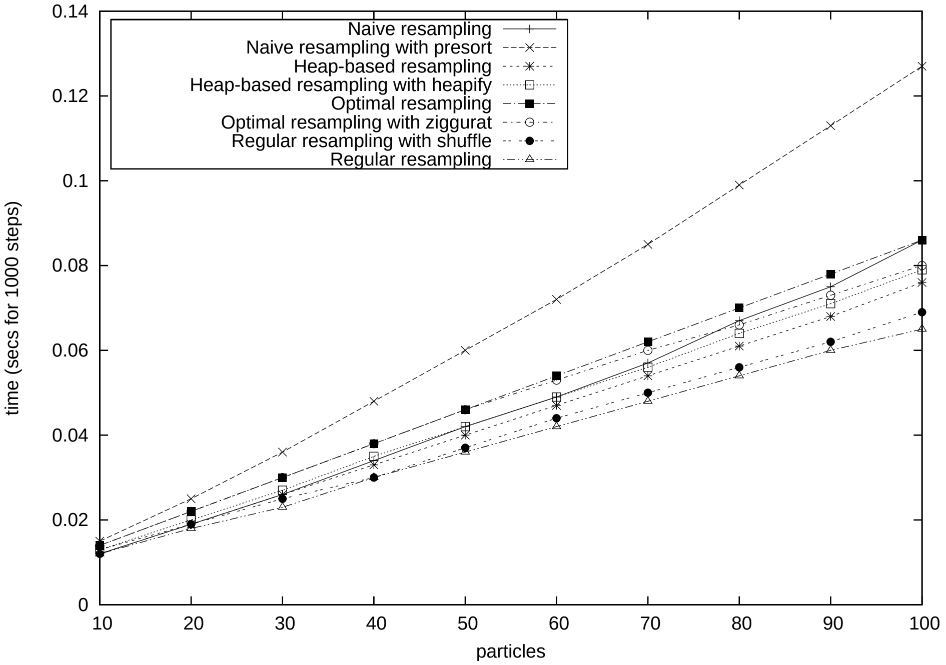


Figure 10: Detail of Figure 9

# 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 [6]. 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 [9], 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 $(1-x)^n$ by some less expensive method than that of Section 2.5. 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 $n$.

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 $n$.

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

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 [3]. 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 do better, though, by modifying the Ziggurat generator to accurately compute the desired power by tweaking the rejection step—this will converge poorly for the last 50 samples or so due to the degenerating approximation, at which point we can just switch to calling `pow()` directly.

#### 4.1.2 Segmentation

The concentration so far has been on generating the uniform variates sequentially. However, for parallelization one would first like to break up the variates into blocks which are treated separately. 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 $\mu_k$ is at position $k$ of $n$, $k-1$ of the samples must have landed to the left of $\mu_k$ 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 $\mu_k$ being the $k^{\text{th}}$ sample is

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

The direct method used in Section 2.5 next requires computing the probability that a variate $\mu$ is in the right position via

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

The integral in the denominator can be calculated directly.

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

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

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

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 $\mu_0$. A rejection method for direction generation of $\mu_0$ 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.

## 4.3 Recommendations

There appear to be three basic regimes that weighted resampling is deployed in. The choice among the algorithms presented here may be largely guided by the deployment.

In the "offline" environment, extremely fast desktop computers or even supercomputers are being used in resampling. In this regime, the cost of floating point is low, the desired resampling accuracy is high, millions or billions of input samples may be presented to the resampler, and parallel processing is often a real option. The perfect optimal algorithm seems like an ideal choice in this domain. One example of this environment is BPF for biomedical signal procesing, with which we have had some involvement.

In the "high performance embedded" environment, a single high-performance CPU, typically without floating point, is available, the desired resampling accuracy is high, real-time performance is usually required, and thousands to tens of thousands of input samples may be preseted to the resampler. In this environment, either our optimal resampler or regular resampling with shuffle may be appropriate. One example of this environment is the BPF sensor fusion on the PPC Linux flight computer aboard our amateur sounding rocket, for which this BPF implementation was originally developed.

In the "low-end embedded" environment, a fairly slow CPU, typically without floating point, is available, the desired resampling accuracy is only moderate, real-time performance is usually required, and hundreds to tens of thousands of input samples may be presented to the resampler. In this regime, a regular resampler, which can be implemented in a very small amount of very fast fixed-point or integer code, seems to be the best match. One example of this environment is a proprietary real-time BPF sensor fusion application for a portable ARM-7 device that we are currently working on.

## 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`.

## 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.

## References

[1] S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for online non-linear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2), February 2002.

[2] 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.

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

[4] D. Fox. Kld-sampling: Adaptive particle filters. *Advances in Neural Information Processing Systems (NIPS)*, 16, 2001.

[5] G. Kitagawa. Monte Carlo filter and smoother for non-Gaussian nonlinear state space models. *Journal of Computational and Graphical Statistics*, 5(1), 1996.

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

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

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

[9] 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.