Flexible Approximate Counting

Scott A. Mitchell
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1316, U.S.A.
samitch@sandia.gov

David M. Day
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1320, U.S.A.
dmday@sandia.gov

ABSTRACT

Approximate counting [18] is useful for data stream and database summarization. It can help in many settings that allow only one pass over the data, want low memory usage, and can accept some relative error. Approximate counters use fewer bits; we focus on 8-bits but our results are general. These small counters represent a sparse sequence of larger numbers. Counters are incremented probabilistically based on the spacing between the numbers they represent. Our contributions are a customized distribution of counter values and efficient strategies for deciding when to increment them.

At run-time, users may independently select the spacing (accuracy) of the approximate counter for small, medium, and large values. We allow the user to select the maximum number to count up to, and our algorithm will select the exponential base of the spacing. These provide additional flexibility over both classic and Csűrös's [4] floating-point approximate counting. These provide additional structure, a useful schema for users, over Kruskal and Greenberg [13].

We describe two new and efficient strategies for incrementing approximate counters: use a deterministic countdown or sample from a geometric distribution. In Csűrös all increments are powers of two, so random bits rather than full random numbers can be used. We also provide the option to use powers-of-two but retain flexibility. We show when each strategy is fastest in our implementation.

Categories and Subject Descriptors

E.4 [**Coding and information theory**]: Data compaction and compression; H.3.3 [**Information Search and Retrieval**]: Information filtering; H.3.4 [**Systems and Software**]: Performance evaluation (efficiency and effectiveness)

General Terms

Algorithms, Performance

IDEAS11 2011, September 21-23, Lisbon [Portugal]
Editors: Bernardino, Cruz, Desai

Keywords

Data stream summarization, approximate counting, random number generator, implementation

1. INTRODUCTION

1.1 Motivation

Summary datastructures are useful whenever the full data are too large to store, such as data streams and out-of-core databases. In monitoring network traffic, a representative application is counting the number of times each pair of IP addresses have communicated. Some pairs communicate a lot, so perhaps we need a 64-bit integer for every pair to store the exact count. That would use up too much memory; besides, for large counts we only care about their order of magnitude. For some database operations, having a sketch of the number of items (or types) allows one to choose a faster algorithm or a more efficient compression scheme.

The "approximate counting" family of methods stores a counter for each item in the stream. This is distinguished from the "heavy-hitter" family of methods, which counts only the most frequent items: e.g. top-k or quantiles [1], and iceberg queries or frequency thresholds [15]. An approximate counter uses less memory than a standard integer counter, but sacrifices some accuracy. It stores about the log of the true count; it is ideal for representing the order of magnitude of the true count. The classic implementation increments a counter probabilistically, using a random number generator.

We describe an implementation of approximate counting, including performance tradeoffs. This module may be customized at compile or run-time, as needed by the application, and for the expected real data.

1.2 Prior Art

1.2.1 Classic Approximate Counting

Classic AC (Morris [18]) is an established method. Each approximate counter stores about the $\log_a$ of the number of observations n; counter value $C \geq 0$ represents the number $N = \phi(C) = (a^C - 1)/(a-1)$. (Here "$a$" is Morris's $(1+1/a)$.) Normally we require $\log_2 M$ bits to store a number as large as M, so storing $\log_a M$ requires only $\log_2 \log_a M$ bits.

The sequence of increasing C defines an increasing sequence of N; the C are consecutive, but the N are not. The counts are inaccurate, i.e. $N \approx n$ only, both because consecutive counts do not represent consecutive integers, and because counts are incremented randomly. If $C = 4$ and

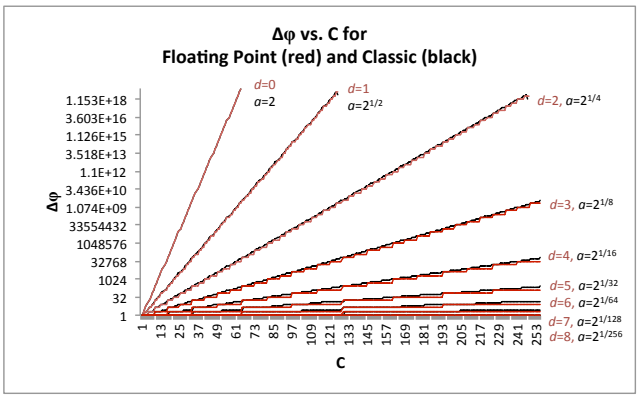

Figure 1: Floating point increments are lower stair-step approximations to classic AC.

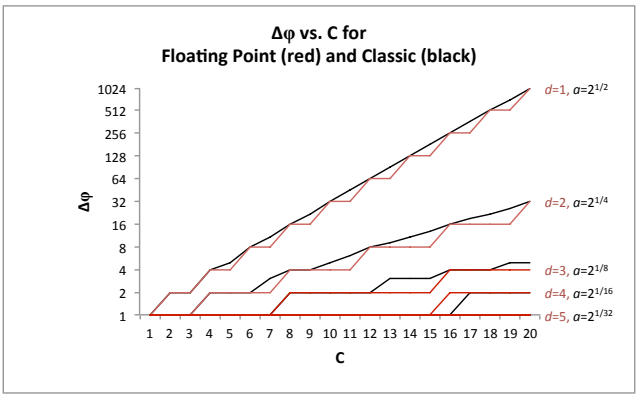

Figure 2: Close-up of Figure 1.

another item is observed, we have only two choices: leave C at 4 or increment it to 5. Given $\Delta\phi(C) = \phi(C+1) - \phi(C)$, we increment C with probability $p = 1/\Delta\phi(C) = a^{-C}$ to ensure the expected value of C is correct.

C is called a *q-ary counter* if $a = 2^{1/r}$ for $r = 2^u$ for integer u. For an 8-bit counter, usually the base a is chosen quite small. $a > 1$ is required but a value of $a = \sqrt[8]{2} \approx 1.09$ might be chosen, since $\phi_{\sqrt[8]{2}}(255) = 3.9\text{e}9$. Note $\phi_2(255) = 5.7\text{e}76$. Kirschenhofer [10] considers a variant of q-ary AC with a factor multiplying the probabilities, as our e in Section 2.1. He analyzes the expected error in detail and contrasts to another counting scheme.

1.2.2 Floating Point Approximate Counting

Recently Csűrös [4] described approximate counting using a binary floating point counter, stored in an unsigned integer. The bits of the counter C are conceptually divided into a d-bit significand u and an exponent t. The significand counts by 1's up to $2^d - 1$; after 2^d the exponent is 1 and the counting increment is 2, etc. In general, $\Delta\phi(C) = 2^t$; and $\phi(C) = \phi(u,t) = (2^d + u)2^t - 2^d$, the value of C when viewed as a binary floating point number.

See Figures 1 and 2. The red curves are the nine possible ϕ functions for Floating Point (FP) AC using 8-bits. FP ϕ are lower stair-step approximations to q-ary counting, base $a = 2^{1/r}$ for $r = 2^u$. FP ϕ functions are piecewise linear, with slope increasing by powers of two. The black curves are the classic case rounded to integer values, as in our Flexible AC. The curves for classic AC with floating point values would be straight lines through the upper red points.

Csűrös's approach has several advantages. Counting is perfect up to 2^d. Since $\Delta\phi$ is always a power of 2, one can use a random bit generator to decide when to increment, which is potentially more efficient than a random uniform floating point number generator. Also $\phi(u,t)$ is fast and easy to calculate.

1.2.3 Error Analysis

Flajolet [6] analyzed the accuracy of approximate counting, and showed that the expected value of the approximate count is correct, and the expected error is acceptable because the variance is bounded. AC error analysis techniques tend to be sophisticated and not easily applied, and often rely on $\Delta\phi$ following a geometric series. Sometimes the base is assumed to be q-ary. Kirschenhofer et al. [9] simplifies Flajolet's analysis, avoiding real analysis by using the calculus of finite differences. Mellin integrals and Euler's formula are central. Later, they [11] extended the analysis technique to many familiar probabilistic algorithms besides approximate counting. Errors are bounded but a limiting distribution of values does not exist. Louchard et al. [14] provides a recent analysis based on generating functions for a Bernoulli model. Csűrös [4] and Kruskal [13] provide nearly identical analysis frameworks. These analyses are straightforward to apply and only rely on approximate counting being a pure birth process (counts only increase) and not the exponential base. We use this to show that our generalized ϕ function has the same asymptotic error as Classic AC in Section 5.

1.2.4 Approximate Counting in Streams

The problem of counting items in a large datastream inspires several classes of algorithms, including approximate counting, probabilistic counting, sampling, and sketches; see "Counting by Coin Tossings" [7] for a survey. Approximate counting is beneficial by itself, and lately it has been combined with the other classes.

Recently AC was blended with sketches. Classic Bloom filters [2] use a single bit to represent the existence of items, regardless of count > 1. The Morris Bloom Counter [5] is a combination of a Bloom Filter with AC: each Bloom filter bucket keeps a counter, as in the Spectral Bloom Filter [3], but the counter is an approximate counter of a few bits. TOB [5, 21] is another way to count using Bloom filters, by layering them, so that a deeper filter represents a more "significant bit". Using a separate hash function for each layer minimizes the consequence of collisions. Deeper layers may use smaller hash tables because they are visited less. The count is unary, e.g. $3 = 1110\ldots$. The TOMB[5] counter replaces the unary bit of TOB with an AC of a few bits; but like unary counting when a layer's counter reaches its maximum it stays at its maximum.

Classic approximate counters use the same number of bits for every counter. For skewed data, this wastes space. Talbot [20] encodes variable-length unary approximate counters for skewed streaming data. A variable number of Bloom-filters select bit locations in shared hash tables. The counters C are unary, the first 0-bit encountered denotes its end. Sampling is blended with AC: frequent items need not visit all their 1-bits every time. He notes ϕ may be an arbitrary

increasing function, chosen for the particular skew.

Flexible AC may be incorporated into any of these.

1.3 Flexible AC Summary

We define ϕ, the counts, and derive p, the probabilities. We think this is the most natural for users. Morris did this in 1978, and Kruskal [13] in 1991, but most authors reverse this for ease of analysis, defining p to follow a geometric series, and deriving ϕ as the expected value.

We store the ϕ function in an array. Our software will function with this array filled with any increasing sequence; Kruskal showed that ϕ may be arbitrary. We think non-decreasing $\Delta\phi$ is a good idea because accuracy is dependent on the largest prior increment. We provide a particular functional form for ϕ we think is an intuitive schema for users, allowing them to customize ϕ to accurately capture the features that are important in their application; see Figure 3. For a fixed number of bits for C, the Classic and Floating Point ϕ functions are dependent on one parameter: the base a or the number of significand bits d. In Floating Point the significand determines a threshold below which the count is perfectly accurate; our users may also set a threshold below which the count is perfect, but this does not determine our entire ϕ function. Users may select the base, as is tradition, or users may select the largest number to count up to, and our algorithms solve for the exponential base. Choosing a base is a common problem for all AC applications, but we have seen no solution method discussed in the literature.

We describe several new options for determining when to increment an approximate counter. Classic AC generates a random number every time an item is seen. One new option is Fixed Countdown: increment after a deterministic number of observations, dependent on the current counter value, but using no random numbers. This works well if the order of items in the stream is sufficiently random to avoid a bias in the counts. A second new option is Random Countdown: precompute a countdown counter equal to the number of times in a row the classic approach would not increment, which is a random number taken from a geometric series. This is more efficient if the probability of increment is small, because most of the time the test simply decrements the countdown. These strategies require a constant amount of extra storage, but can dramatically reduce how many random numbers we need to generate, improving the run-time.

In Csűrös's Floating Point AC all increments are powers of two, 2^k. He can test whether to increment by generating up to k random bits, stopping when the first non-one bit is generated. We can round the numbers our counter represent to also follow powers-of-two increments, but retain ϕ flexibility. We achieve about the same run-time efficiency as Csűrös, with a constant-factor of memory overhead to store ϕ in an array.

In Classic AC the numbers $N = \phi(C)$ are floating point. Reporting to a user that an item had been observed "4.28" times might be confusing, although it would be a reminder that the count is only approximate. In our implementation, as in Floating Point, our ϕ values are always integers, although this is not fundamental except when we are using power-of-two increments.

We provide an experimental comparison of the run-time efficiency of all of these strategies. Fixed Countdown is by far the fastest. If randomization is required, we think it is best to use Classic or Csűrös's RandomBit decisions for small increment values then switch to Random Countdown for large increment values.

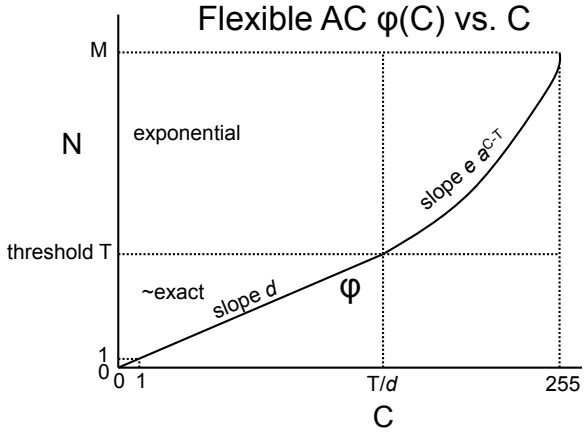

Figure 3: Functional form of ϕ for our Flexible AC.

2. FUNCTIONAL FORM

2.1 Flexible Phi

Our ϕ function is a discrete integer, strictly increasing version of the continuous, strictly increasing function Φ.

$$\Delta\Phi(C) = \begin{cases} 1 & \text{if } C = 0 \\ d & \text{if } 1 \le C < T \\ ea^{C-T+1} & \text{if } C \ge T \end{cases}$$

$$\Phi(C) = \begin{cases} 0 & \text{if } C = 0 \\ 1 + d(C-1) & \text{if } 1 < C \le T \\ 1 + d(T-1) + e\frac{a^{C-T+1}-a}{a-1} & \text{if } C > T \end{cases}$$

Where $a > 1$, $d \ge 1$, $e > 0$, and $T \in \mathbb{N}$. Classic counting is $T = d = e = 1$. Here $\Delta\Phi = \Phi(C+1) - \Phi(C)$ and Φ follows from $\Delta\Phi$ through the identity $\sum_{i=1}^{K} a^i = (a^{K+1} - a)/(a-1)$, for $a > 1$. For $C > T$, we compute $\phi(C) = \phi(T) + \lfloor ea^{C-T} \rfloor$. This ensures $\Delta\phi$ is non-decreasing. We enforce $\Delta\phi \ge 1$. Given ϕ, we calculate $p(C) = 1/(\phi(C+1) - \phi(C))$. Both ϕ and p are stored in arrays for speed. Finding C given N can be done using binary search on the array.

2.2 Powers-of-Two Flexible AC

When constructing ϕ, we have the option of rounding (up and down) all the increments to the nearest power-of-two. Csűrös showed experimentally that the loss of accuracy is insignificant. Figure 4 illustrates that our method can produce ϕ curves between the choices that a pure floating point counter can provide. E.g. consider the large open wedges in Figure 1 between $d = \{1, 2, 3\} \iff a = \{\sqrt{2}, \sqrt[4]{2}, \sqrt[8]{2}\}$.

3. SOLVING FOR THE BASE IN AC

We find a so that $\Phi(\max(C)) = M$. Recall ϕ is a discrete approximation to Φ, so $\phi(\max(C)) \ne M$ exactly. However, we usually achieve $\phi(\max(C)) \approx M$ within many significant digits, and only order-of-magnitude accuracy is needed.

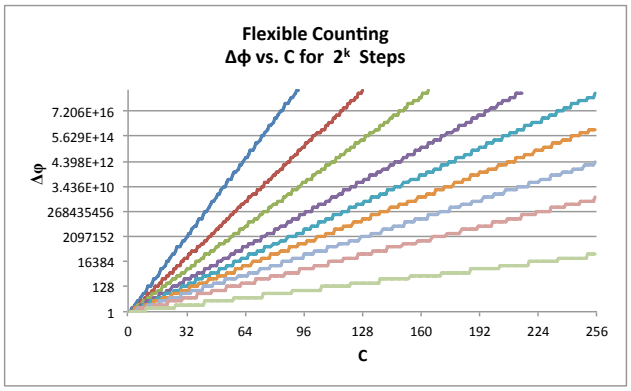

Figure 4: ϕ for power-of-two increments, $\log_2$ scale.

Determining a given $\max(C), M, d, e$, and T resolves to an inverse geometric sum problem. Classic AC is just the special case of $d = e = T = 1$. $\Phi(T)$ is determined by T and d, independent of a and e. For an 8-bit counter, $\max(C) = 255$, but here it can be arbitrary. Let $K = \max(C) - T \geq 1$.

$$M = \Phi(\max C) = 1 + d(T-1) + e\frac{a^{K+1} - a}{a - 1} \tag{1}$$

$$\iff r(a) = a^{K+1} - sa + (s-1) = 0, \tag{2}$$

where $s = (M - 1 - d(T-1))/e + 1$. We seek the largest positive real root of $r(a)$ (2), since $a = 1$ is also a root but not a solution to the original problem (1).

We have a reliable and efficient algorithm. The routine has been tested for $1 \leq K \leq 1000$ and selected floating point values of s up to the range of numbers representable in IEEE arithmetic. The equation is solved in constant time. The root that is approximated is the largest root. The subtle points are the initial guess, the choice of nonlinear solver, and the convergence threshold.

3.1 Initial Guess

A good initial guess a_o for a depends on whether s is small, medium or large relative to K. *Small* is $s - 1 \in [0, K + 1]$, *medium* is $\in [K + 1, ((4K)/(K-1))^{K-1}]$, and *large* is $\in [(4K)/(K-1))^{K-1}, \infty]$. For small s, the initial guess is the first order expansion of a in $s - K$.

$$\text{Small } s\text{: } a_o = 1 + \frac{2(s-K)}{K(K+1)}.$$

Medium s is both the most common and the most difficult case to derive. Define $q(a) = \sum_{k=1}^{K} a^k$. A solution to (1) has $q(a) = s - 1$, at which point $\frac{da}{ds} = 1/q'(a)$.

$$\int_{q^{-1}(K)}^{q^{-1}(s-1)} da = \int_K^{s-1} \frac{da}{ds} ds = \int_K^{s-1} \frac{q(a)}{q'(a)} \frac{ds}{s} \tag{3}$$

Since $\frac{q(a)}{q'(a)} > 2a/(K+1)$ and approaches equality as $a \to 1$, we simplify (3) using $\frac{q(a)}{q'(a)} \approx 2a/(K+1)$.

$$\approx \frac{2a}{K+1} \int_K^{s-1} \frac{ds}{s} = \frac{2a}{K+1} \ln \frac{s-1}{K} \tag{4}$$

On the left of (3) we have $\int_{q^{-1}(K)}^{q^{-1}(s-1)} da = q^{-1}(s-1) - q^{-1}(K) = a(s-1) - 1$. Combined with the right of (4) we have our initial guess $a_o = a(s-1)$.

$$\text{Medium } s\text{: } a_o = 1 + \frac{2}{K+1} \ln(\frac{s-1}{K}).$$

For large s our initial guess is an approximation to the largest positive root of $y^{K+1} - sy + s = 0$. This equation is equivalent to $y = y_o(\frac{y-1}{y})^{1/K}$ for $y_0 = s^{1/K}$. We take y_j as our initial guess, by doing a couple of fixed point iterations.

$$\text{Large } s\text{: } a_o = y_4 : y_0 = s^{1/K},\ y_{j+1} = y_o \left(\frac{y_j - 1}{y_j}\right)^{1/K}.$$

3.2 Solver

Given any of these initial guesses, we solve for a using a nonlinear solver. In practice, the initial guess is very good and only a few (about 3) solver iterations are needed. The basic nonlinear solver is Euler's method [22]. Euler's method uses a second order expansion, and selects as the approximate solution the root of smallest absolute value. A Newton update is used if the roots of the quadratic are complex. Euler's method is globally and monotonically convergent in exact arithmetic [22]. A second order method also handles the double root at $s = K$ seamlessly. A special form of recurrence formula is used if the absolute value of the residual is greater than the square root of the overflow threshold.

The convergence threshold for approximate roots of $r(a)$ (2) is proportional to the error in evaluating $r(a)$ in floating point arithmetic, namely $\epsilon(1 + K/4)\max(|r'(a)|, a^k - sa + (s-1))$, where ϵ is machine precision.

3.3 Phi With Easy Inverse

We are tempted to dispense with the need for a solver by choosing a Φ with an easy inverse. One choice is the sum of a linear and exponential function: $\Phi_{\text{easy}} = d(C-1) + ea^{C-1}$ for $C > 1$, with $T = 1$. This yields $a = \sqrt[K]{r}$ where $K = \max(C) - 1$ and $r = (M - dK)/e$. This function is about the same as our Φ_{FAC} for large C, but is much smaller for small C.

4. STRATEGIES FOR INCREMENTING

We suggest a few ways to speed up deciding whether to increment C. Figure 5 shows the main datastructures.

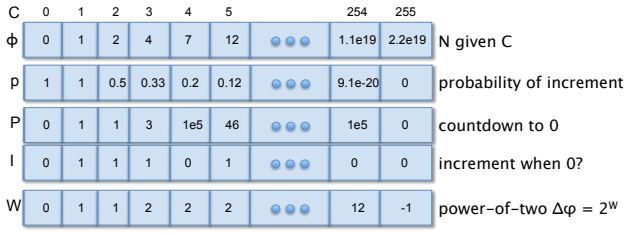

Figure 5: The main datastructures for flexible AC.

4.1 Classic AC Speedup

As a trivial warm-up, note that classic AC can be sped up using a fast RandomBit algorithm, even when the probabilities are not inverse powers of two, using the ideas of Csűrös [4]; see Algorithm 1. The expected time is reduced if generating a random double takes at least twice as long as generating a RandomBit. In our case the time-savings is

Algorithm 1 Decide to increment, speedup Classic AC.

```
while p ≤ 0.5 do
  if RandomBit then
    return 0; [do not increment C]
  end if
  p ← p × 2.
end while
[now test as Classic AC]
return (p = 1 or random_double < p)
```

about 20% for general $\Delta\phi$. The time in Figure 7 does not include this speedup.

4.2 Countdown

A drawback of Classic AC and Floating Point AC is that the generation of random numbers or bits can be compute intensive, at least compared to incrementing an integer! We introduce the idea of using a countdown counter P instead. Every time a key is seen, we decrement P, and when we reach zero we increment C and reset P. If the probability of increment is small, the countdown is likely large, so most of the time we are simply decrementing.

If we had one countdown counter per item we are counting, then this takes up as much memory as normal counting, or more. However, we only need one countdown counter per C *value*. This only adds a small constant amount of memory, an array of the same size as our ϕ values.

This method can reproduce the behavior of classic approximate counting, which shows it is provably correct: set the value of P by actually generating random numbers until one smaller than the probability of increment was generated, set P to the number of random numbers generated. We speed this up, achieving both correctness and efficiency in Random Countdown. We also provide a deterministic variation. See Section 4.4 for how the run-times compare.

4.2.1 Random Countdown

This method has identical expected behavior to Classic AC, subject to the limits of random number generators. P_{classic} is the number of times in a row $u > p(C)$, where u is a uniform random number from $[0, 1]$. This is the definition of a geometric distribution. Generate a random number u uniformly from $[0, 1]$, then $P = \lfloor(\ln(u)/\ln(1 - p(C))\rfloor$ has the same distribution as P_{classic}.

4.2.2 Hybrid

Hybrid uses Classic AC for $\Delta\phi \leq 10$, then switches to Random Countdown. Hybrid could also be combined with RandomBit, both Floating Point and Powers-of-two, switching for $\Delta\phi$ about 18–22.

4.2.3 Fixed Countdown

The countdown value is set to a preset value. This is the most efficient method. If there is only one item to count, it is also the most accurate. However, for multiple items, the approximate counts will be unbiased only if the order of the items in the stream are random. For example, if two items alternate in the stream, one will likely be undercounted and the other overcounted. For any fixed ϕ, certain stream patterns would generate extreme inaccuracy.

A naïve choice for the countdown value is the reciprocal of the increment probability, $P(C) = \Delta\phi(C)$. But this deterministically undercounts, so the expected value is incorrect. Consider counting a single item: we want to report a value of $\phi(C)$ when n is closest to $\phi(C)$. A good choice is $P(C) = (\phi(C + 1) - \phi(C) + 1)/2 + (\phi(C) - \phi(C - 1))/2$, where division denotes computer division on integers so that fractional values are dropped, and $\phi(-1) = 0$. (Note that $P(C) = (\phi(C + 1) - \phi(C - 1) + 1)/2$ will be incorrect if $\phi(C + 1) - \phi(C - 1)$ is consistently odd or even.) Figure 6 illustrates the concept and gives example values. If only one item is approximately counted, then $N = \phi(C)$ is reported for true count $n \in [n\min, n\max]$.

C	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Δφ	1	2	3	4	5	6	7	8	9	10	11	12	13	
P	1	1	3	3	5	5	7	7	9	9	11	11	13	
n min	0	1	2	5	8	13	18	25	32	41	50	61	72	
φ	0	1	3	6	10	15	21	28	36	45	55	66	78	91
n max	0	1	4	7	12	17	24	31	40	49	60	71	84	

C	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Δφ	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	
P	1	1	3	6	12	24	48	96	192	384	768	1536	3072	
n min	0	1	2	5	11	23	47	95	191	383	767	1535	3071	
φ	0	1	3	7	15	31	63	127	255	511	1023	2047	4095	8191
n max	0	1	4	10	22	46	94	190	382	766	1534	3070	6142	

Figure 6: Two fixed countdown examples.

4.3 Numerical Issues

There are some issues with very small numbers, both representing them and generating them. In the classic case, we must generate a random number u uniformly from $[0, 1]$ and check if $u < p(C)$. The value $p(C) = 1/\Delta\phi(C)$ might be quite small, say 10^{-10}. It is important to know the limits of the random number generator, the smallest value t such that the probability of generating a value less than t is approximately t, $p_{\text{rand}}(u < t) \approx t$ We cannot distinguish between $u < t$, and should treat them all as some indeterminate value less than t. If $p < t$, then we cannot determine whether we should increment by generating a single random number. Instead, since $p = p_{\text{rand}}(u_1 < t)p_{\text{rand}}(u_2 < p/t)$, we generate u_1, and if it is less than t, then we generate a second number and test if it is less than p/t. (Repeated until $u_i \geq t$.)

Several numerical issues arise in the random countdown strategy. We are expanding a distribution over $[0, 1]$ to a distribution over $[1, \infty)$ so gaps between the distribution of generated floating point numbers are exaggerated. For some applications this may introduce an unacceptable bias. We take care to fill in the gaps near 0 and 1, necessary for extreme p. If $u < t$ we compute P based on $v = t$ and set a flag so that when the countdown counter reaches zero, we do not increment C, but just reset P to some new random value; see the "increment when 0?" binary array in Figure 5. If $u > T$, where T is the largest value such that $p(u > T) \approx 1 - T$, then (depending on p) we must treat $u_1 = T$ and generate subsequent numbers to improve the accuracy for small P.

4.4 Efficiency Comparison

We implemented and tested each of the strategies. Figure 7 plots run times for increments < 64. For increment values < 256, we tested whether to increment a billion (1e9) times, keeping the increment value constant throughout the test. For RandomBit, the increment is rounded to the near-

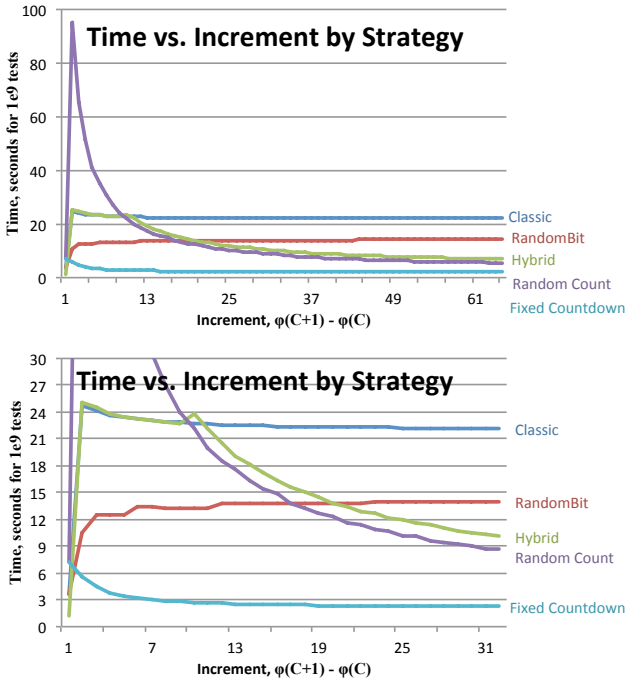

Figure 7: Time to test whether to increment C vs. the increment (often $1/p(C)$) for different strategies.

est power of two, hence the stair-step curve. RandomBit is used in Floating Point AC and Powers-of-two AC. Hybrid switching requires a small amount of overhead, which is why the Hybrid curve lies above Random Countdown. The time on the very left of Figure 7 reflects overhead, because if the probability of increment was 1 no random number was generated. Re-running the experiments at different times resulted in a variation of about 5% due to machine activity. In practice running times will depend on factors such as the random bit and floating point sources, and the application context, but we observe these trends.

Deterministic countdown is the fastest, as it involves simply decrementing a countdown-counter; if it reaches zero there is a little extra work to reset it from a lookup table, and to increment the actual counter. The extra work happens less frequently as the increment (countdown value) increases. Classic counting requires generating a random floating point number at each check. There is a little extra work if the number is incremented. Hence the time goes down very slightly as the probability decreases. In contrast, RandomBit generates a variable number of random bits. The time increases slightly as the probability decreases, i.e. the maximum number of successive bits to check increases. The procedure to set the Random countdown counter is very costly, involving computing two (approximate) logs, and several underflow/overflow checks. This gets done more frequently for small increments (large probabilities). However, Random countdown is otherwise the same cost as deterministic countdown. This is why we see Random countdown having terrible runtime for small increments, but eventually beating every other strategy besides Deterministic Countdown.

The times spanned several orders of magnitude. RandomBit was about twice as fast as Classic, with the biggest win for an increment of 2. Random Countdown was about the same as RandomBit for increments of about 20, becoming increasingly better, e.g. by a factor of 4 when the increment was 172. Random countdown took only 1.5 times as long as deterministic countdown when the increment was 255.

4.4.1 Random Number Generators

For the source of random bits, we used Equation 10 in Knuth's vol. 2 [12], with 64-bit integers. Besides the seed, the method also accepts a constant 64-bit input parameter A, representing the coefficients of a primitive polynomial. We set A to have a 1-bit in positions 56, 49, 40, 31, 24, 16, 8, and 0, because Knuth hints that a polynomial with more non-zero coefficients is better, and Rajski and Tyszer [19] describe this as a degree-64 primitive polynomial with many 1-coefficients. This provided a very good distribution of bits for our requirements; the sequences of consecutive 1's occur with about the expected frequency as a truly random source. By generating about 1e14 bits we verified the distribution's desired behavior up to 42 consecutive 1's.

We also experimented with generating random integers using Marsaglia's KISS [16] and SUPRKISS64 [17], then extracting bits as needed. This took about 20% longer than Equation 10 with no apparent advantages.

For the floating point random numbers, we used a Fibonacci generator with about 100 state variables. We used a variation of RANMAR from James [8], which is itself a clean-up of an earlier code by Marsaglia and Zaman.

4.4.2 Recommendations

If speed is important, use Deterministic Countdown if the stream is sufficiently random. Otherwise, use RandomBit or Classic AC for small C, and Random Countdown thereafter. Use RandomBit rather than Classic if stairstep $\Delta\phi$ is acceptable. In our implementation "small C" means switching at $\phi(C+1)-\phi(C) = 10$ for Classic or 22 for RandomBit. The threshold in other implementations will depend on e.g. the complexity and accuracy of the underlying random number generators; compiler optimizations; how the overhead of the checking strategy is implemented; and the hardware, e.g. floating vs. integer flop speed, memory cache. If a lot of items are expected to be observed only once, make sure initializing the count is fast.

5. ACCURACY

We analyze our accuracy using the straightforward analytic framework of Csűrös [4]. Define *accuracy* to be $A_n = \sqrt{\text{Var } \phi(X_n)}/\mathbb{E}\phi(X_n)$, where Var is variance and $\mathbb{E}$ is expected value. Viewing AC as a Markov chain, the X_n are the random-variables (C-value states) after counting n items. We restate a key theorem using our notation.

THEOREM 1 (CSŰRÖS THEOREM 2).

$$\text{Var } \phi(X_n) = \mathbb{E}g(X_n), \textit{where } g(C) = \sum_{c=0}^{C-1} \frac{1}{p^2(c)} - \frac{1}{p(c)}$$

LEMMA 1.

$$g(C) = \begin{cases} 0 & \textit{if } C \le 1 \\ (C-1)(d^2-d) & \textit{if } 1 < C \le T \\ g(T) + e^2\frac{a^{2D}-a^2}{a^2-1} - e\frac{a^D-a}{a-1} & \textit{if } C > T \end{cases}$$

where $D = C - T + 1$.

Csűrös's Theorem 3 provides our asymptotic accuracy. $\lambda^2 = \lim_{C\to\infty} \frac{g(C)}{\phi^2(C)} = \frac{a-1}{a+1}$. Then $\lim_{n\to\infty} A_n^2 = \lambda^2/(1-\lambda^2) = (a-1)/2$. This is the same as in Classic AC [6] and Floating Point AC [4], as our extra e terms cancel, and our other extra terms are unimportant in the limit.

6. CONCLUSIONS

In summary, we provide convenient control parameters to generate ϕ functions customized for users' applications. Asymptotic errors are similar to classic approximate counting. We have given the first description of how to determine the exponential base for (flexible) approximate counting given a desired maximum count, and an alternative that requires no solver. The community might explore alternate functions that have both an easy inverse and the desired accuracy in the regimes of interest.

We have shown how to speed up the process of deciding when to increment; some datastream applications need to be very fast. We have shown that using a countdown counter can be much faster than even RandomBit decisions when the probability of increment is small. All of our implementation results used an 8-bit counter, but the techniques generalize.

In the future, those using approximate counting should consider this flexibility to tailor their functions for accuracy in the regime they care about. Error rates for these regimes should be studied empirically in the context of the intended application, as needs and consequences vary widely. Implementers should consider how increment-decisions affect run-time. When selecting a random number generator or random bit generator, one should consider how it is used in approximate counting; its useful limits may be different than what other tests of "randomness" might imply.

Acknowledgments

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

7. REFERENCES

[1] Arvind Arasu and Gurmeet Singh Manku. Approximate counts and quantiles over sliding windows. In *Proc. Symp. on Principles of Database Systems*, PODS '04, pages 286–296. ACM, 2004.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[3] Saar Cohen and Yossi Matias. Spectral Bloom filters. In *Proceedings International Conference on Management of Data*, SIGMOD '03, pages 241–252. ACM, 2003.

[4] Miklós Csűrös. Approximate counting with a floating-point counter. In M. T. Thai and S. Sahni, editors, *COCOON*, volume 6196 of *Lecture Notes in Computer Science*, pages 358–367. Springer, 2010.

[5] Benjamin Van Durme and Ashwin Lall. Probabilistic counting with randomized storage. In *Proceedings International Joint Conference on Artificial Intelligence*, IJCAI, pages 1574–1579, July 2009.

[6] Philippe Flajolet. Approximate counting: A detailed analysis. *BIT*, 25(1):113–134, 1985.

[7] Philippe Flajolet. Counting by coin tossings. In Michael J. Maher, editor, *ASIAN*, volume 3321 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2004.

[8] F. James. A review of pseudorandom number generators. *Computer Physics Communications*, 60:329–344, October 1990.

[9] Peter Kirschenhofer and Helmut Prodinger. Approximate counting: An alternative approach. *ITA*, 25:43–48, 1991.

[10] Peter Kirschenhofer and Helmut Prodinger. A coin tossing algorithm for counting large numbers of events. *Mathematica Slovaca*, 42(5):531–545, 1992.

[11] Peter Kirschenhofer, Helmut Prodinger, and Wojciech Szpankowski. Analysis of a splitting process arising in probabilistic counting and other related algorithms. *Random Struct. Algorithms*, 9(4):379–401, 1996.

[12] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley, 1998.

[13] J. Kruskal and A. Greenberg. A flexible way of counting large numbers approximately in small registers. *Algorithmica*, 6:590–596, 1991.

[14] Guy Louchard and Helmut Prodinger. Generalized approximate counting revisited. *Theor. Comput. Sci.*, 391(1-2):109–125, 2008.

[15] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings International Conference on Very Large Data Bases*, VLDB '02, pages 346–357, 2002.

[16] George Marsaglia. 64-bit KISS RNGs. velocityreviews website, 28 Feb 2009, `http://www.velocityreviews.com/forums/t673657-64-bit-kiss-rngs.html`.

[17] George Marsaglia. SuperKISS for 32- and 64-bit RNGs in both C and Fortran. math.sci newsgroup posting, 27 Nov 2009, `http://groups.google.com/group/sci.math/msg/af781ad30191a4fe`.

[18] Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

[19] Janusz Rajski and Jerzy Tyszer. Primitive polynomials over GF(2) of degree up to 660 with uniformly distributed coefficients. *J. Electronic Testing*, 19(6):645–657, 2003.

[20] David Talbot. Succinct approximate counting of skewed data. In *Proceedings International Joint Conference on Artificial Intelligence*, IJCAI, pages 1243–1248, July 2009.

[21] David Talbot and Miles Osborne. Randomised language modelling for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 512–519, June 2007.

[22] Joseph Frederick Traub. *Iterative methods for the solution of equations.* Prentice-Hall, 1964.