Editors: Francis Sullivan, fran@super.org
William J Thompson, wmjthompson@msn.com

PAY ME NOW OR PAY ME LATER

Isabel Beichl and Francis Sullivan

A COLLEAGUE RECENTLY ASKED FOR A WAY TO SAMPLE FROM A POISSON DISTRIBUTION TO SIMULATE THE WAY METALLIC FLAKES CLUSTER IN PAINT (SEE FIGURE 1). THE FLAKES AREN'T DISTRIBUTED UNIFORMLY AT RANDOM THROUGHOUT the paint binder but form clusters around center points. The number of flakes around a center is distributed like the Poisson distribution. In this installment of "Computing Prescriptions," we give three ways to sample from the Poisson distribution. We describe the usual method, a faster method that uses some preprocessing, and an even faster method with more preprocessing. The last two methods work with any discrete probability distribution. Depending on how often we need to generate samples, paying extra for a deluxe model of sample generation might be worth it.

The standard method

A typical way to generate samples from a Poisson distribution is to specify an integer μ (the mean of the samples will be μ) and generate random uniformly distributed numbers, $0 < u < 1$, until the product of all those chosen is less than $e^{-\mu}$. Here is pseudocode:

```
mu is given.

u = random
k = 1
while(u > e^(-mu))
  u = u*random
  k = k + 1
endwhile
k has now been selected
from the Poisson distribu-
tion.
```

This looks somewhat like a Metropolis "rejection" method, but it isn't, because we multiply the random numbers together. However, it is similar in that we generate random numbers and then "reject" them and generate more. On the average, it takes μ iterations of the `while` loop before reaching the accepted k. If μ is large and we needs lots of k's, this might mean a *long* time to wait. We don't like that at all. On top of that, μ is an average waiting time: the actual number of iterations can be much larger. So, we'd like to have a way to sample from this distribution without using the loop.

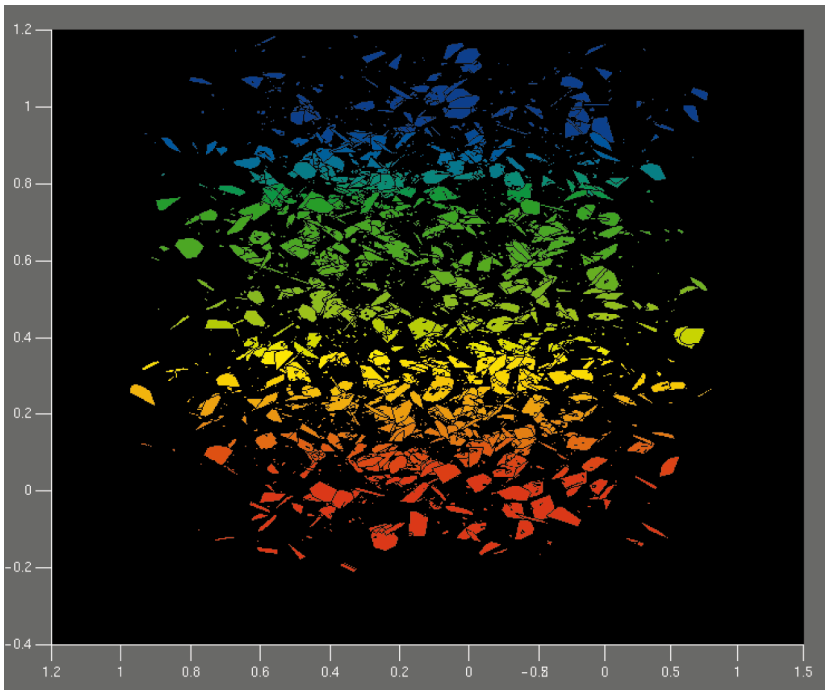

Figure 1. Simulation of metallic paint flake cluster size distributed like Poisson, illustrating work by Isabel Beichl and Fern Hunt at NIST.

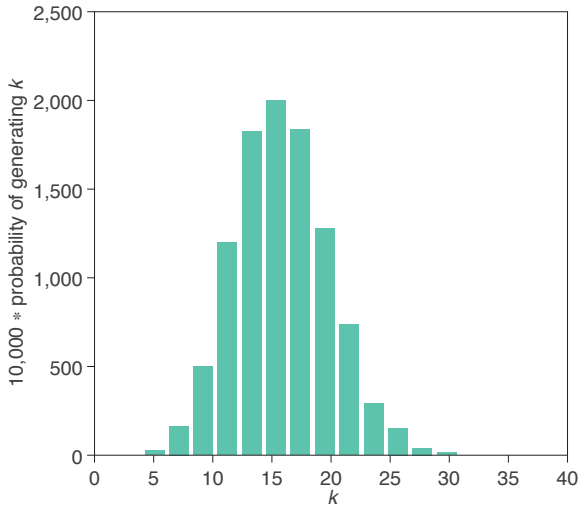

Figure 2. A bar graph of a Poisson distribution, $\mu = 15$.

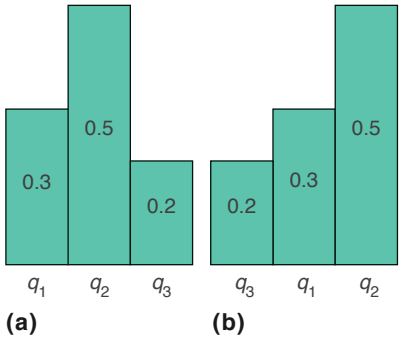

Figure 3. Bar graphs of a very simple discrete probability distribution: (a) the original; (b) sorted.

A faster method

We can speed things up by doing a little preprocessing. First, make an array q_k of the probability of choosing k. One way to do this is to look these values up in a table; another way is to generate a huge number of samples and make your own; another way in the case of the Poisson distribution is to use the formula

$$q_k = e^{-\mu} (\mu^k / k!).$$

Why this is the same as the pseudocode is an interesting story for another occasion. Figure 2 shows a bar graph of the probabilities of choosing various k's for a Poisson distribution, but this method works as long as you can get some array of probabilities.

Once we have an array of q's, we make an array of partial sums where s_k is the sum $q_1 + \ldots + q_k$. That is,

$$s_1 = q_1$$
$$s_1 = q_1 + q_2$$
$$s_2 = q_1 + q_2 + q_3$$
$$\ldots$$
$$s_n = q_1 + \ldots + q_n.$$

Then, to select from this distribution, select uniformly at random between 0 and s_n and do a binary search into the s_n's. The binary search's complexity is only $O(\log n)$. The random number will "hit" one of the s's, and the corresponding k is the sample.

The deluxe method

Our deluxe method comes from Donald Knuth's *The Art of Computer Programming, Vol. 2*, and is called the *alias method*.[1] As in the previous method, we need a discrete probability distribution in an array, q, and we will do some preprocessing. Figure 3a shows an example. For simplicity, we suppose that only three states are possible, with probabilities 0.3, 0.5, and 0.2. Then we sort the q's but remember to keep track of where the sorted values came from. Figure 3b shows the bars of Figure 3a after sorting, where $loc[i]$ contains the original location of bar i. So $loc[1] = 3$ agrees with the picture where bar $q[3]$ is now first.

We then make *boxes* all of length $1/n$, where n is the total number of q's available. We make n boxes all together so that the total length will be 1. So, for our example of three bars, we make three equal-size boxes of length 1/3. In the first box, we put the smallest (that is, first after sorting) bar from the q array. This bar will always fit entirely in a box of size $1/n$. This is because the largest the first bar could be is $1/n$, in the case where all bars are the same size, there being n bars all adding up to 1. We then fill the rest of the box, using part of the bar from the largest bar. By the same reasoning, the smallest that the largest bar could be is $1/n$. This is because, as before, the worst case is the one where all bars are the same size, which would make them $1/n$. Thus, no matter how small the first bar is, the last bar will have enough left to fill the first box. Figure 4a illustrates making the first box from the smallest and largest bars. The boxes are horizontal to distinguish them from the bars. For efficiency, we index this box by the index of the original location of the bar that goes at the box's beginning.

To make the second box, we remove the first bar (which we've already put in a box) and the part of the last bar that was used (see Figure 4b). We're now in the same position as when we started. We sort the remaining bars, put the smallest in box 2, and take part of the largest bar to fill the rest of the box. We continue in this way until we've put all the bars into boxes.

We associate two numbers with each box: a break point p and an alias Y. The break point is the length of the first bar in the box. So, in Figure 4a, the break point for box 1 is $p[1] = 0.2$ because bar $q[3]$, the first part of the box, has a length of 0.2. We know that $q[3]$ is the corresponding bar because we know $loc[1] = 3$, this bar's position before the sorting.

That's the preprocessing. In the pseudocode (see Figure 5), we multiply the box size by n because it will make generating an actual sample slightly easier. It doesn't change the idea of the method. Imagine only that there are n boxes, each of size 1.

Here's how to generate a sample. We select a box by choosing an integer between 1 and n,

$$u = random,\ k = \lceil n * u \rceil,$$

and then find its fractional part

$$v = mod(u, 1).$$

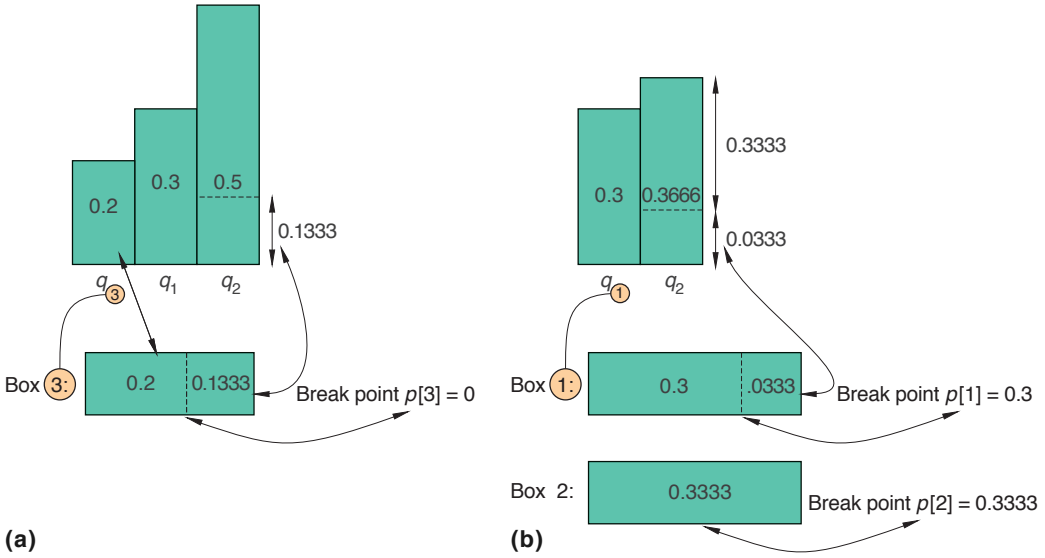

Figure 4. Making boxes from the bars in Figure 3: (a) the first box; (b) the other boxes.

Choosing k selects a box. We use v to select either the first bar in the box or the alias. That is, if $v < p_k$, we use box number k; otherwise, we use box Y_k. We don't even need to choose two random numbers. v is a random number between 0 and 1, so it is the same size as the box. This is why we multiplied the box size by n in the pseudocode.

Here's the pseudocode for the sample generation:

```
u = random
k = ceil(n*u)
v = mod(n*u, 1)

IF v < p[k] THEN
    sample = k
ELSE
    sample = Y[k]
ENDIF
```

Figure 6 shows for our simple example how, if the random number we select happens to be 0.9, multiplying by 3 and taking the ceiling gives $\lceil 2.7 \rceil = 3$ so that box 3 is selected. And within that box, $v =$

```
make pdf[.]                     /* probability distribution function */

kk = n                          /* kk = loop counter, n = size of distribution */

[q, loc] = sort(pdf,1,n)        /* q is the sorted array, loc are the locations from whence the
                                   new values came, sort the whole array from positions 1 through n*/
ismall = 1;                     /*array to be sorted goes from q[ismall] to q[ilarge] */
ilarge = n;

WHILE (kk > 0)
  ix = loc[ismall]              /*index of next box to be filled*/
  p[ix] = n*q[ismall]           /* break point: length of first part of box = smallest bar*/
                                /*multiply by n because it will make stabbing easier, box size
                                consistently n times larger*/
  Y[ix] = loc[ilarge]           /* original location of bar in latter part of box */
                                /*the alias */
  q[ilarge] = q[ilarge] - (1/n - q[ismall])      /*adjust what's left in large bar by subtracting
                                                   what went into box*/
  ismall = ismall + 1
  [q,newloc] = sort(q,ismall,ilarge)            /*sort only between ismall to ilarge*/

  loc[.] = loc[newloc[.]]       /*loc[i] = loc[newloc[i]], what is now in location i was previously
                                in newloc[i], which was previously in loc[newloc[i]]*/
  kk = kk - 1
ENDWHILE
```

Figure 5. Pseudocode for preprocessing in the deluxe method.

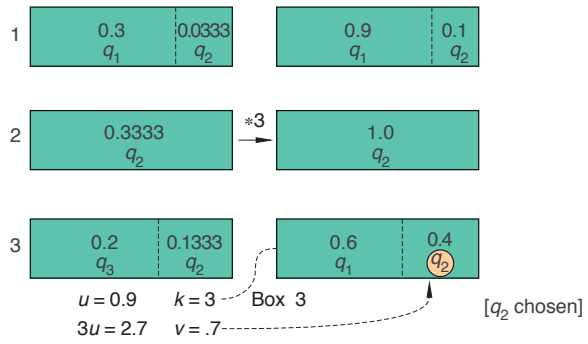

Figure 6. Generating a sample with the deluxe method.

$2.7 mod 1 = 0.7$, which is greater than the break point, 0.6. So, we choose the alias for box 3, which is event 2.

So what do you pay for these? It depends on how big μ is for the Poisson distribution and how many samples S you intend to generate. The traditional method for generating samples from a Poisson requires, on the average, μ random numbers per sample. So it will use on the average $\mu * S$ random numbers, which could get expensive for large μ and huge S.

The faster method has an initial cost of $O(n)$ for making partial sums, where n is the number of bars in the bar graph. Here n can be, on average, around 3μ for the Poisson. To generate the samples, however, will cost $O(S * \log n)$ because a binary search costs $O(\log n)$.

The deluxe method initially will cost $O(n \log n)$ for the first sort. Later sorts really involve only inserting one element into the list, which could be done with a heap at a cost of $O(\log n)$ per box.[2] For n boxes, this would be another $O(n \log n)$. But, to run one sample now only uses one random number and is an $O(1)$ operation. S samples cost S random numbers. So, when it comes right down to it, what would any of the preprocessing matter if S is equal to a billion? Whether or not you want to pay the up-front costs will depend on the problem. Sometimes it pays.

References

1. D.E. Knuth, *The Art of Computer Programming, Vol. 2*, Addison-Wesley, Reading, Mass., 1998.
2. I. Beichl and F. Sullivan, "A Heap of Data," *IEEE Computational Science & Eng.*, Vol. 3, No. 2, Summer 1996, pp. 11–14.

Isabel Beichl is a mathematician in the Information Technology Laboratory at the National Institute of Standards and Technology. Contact her at NIST, Gaithersburg, MD 20899; isabel@cam.nist.gov.

Francis Sullivan is the associate editor-in-chief of *CiSE* and director of the Institute for Defense Analyses' Center for Computing Sciences. Contact him at the IDA/Center for Computing Sciences, Bowie, MD 20715; fran@super.org.