How to Make a Lumpy Random-Number Generator

Michael A. Covington

Institute for Artificial Intelligence
The University of Georgia
mc@uga.edu

ABSTRACT

Normally, random-number generators are designed to produce numbers with a uniform distribution. The sum of uniform random variates has a bell-curve-shaped distribution. Using bell curves like wavelets, individual uniform random variables can be summed to produce arbitrary nonuniform distributions. The result is a simple, customizable non-uniform random-number generating algorithm that has been prototyped on Plan 9 and is equally suitable for other computing environments, including very small embedded systems.

1. Problem definition

Normally, random-number generators produce uniformly distributed values. However, nonuniform random numbers are needed for a number of purposes. In simulation, one often needs random numbers conforming exactly to the observed or theoretical distribution of an input variable, in order to produce an authentic distribution of simulated output [1].

In other situations, the requirements are much less precise, but random numbers with a preference for certain values or ranges are still desired. Examples include equalizing wear on machinery by having a machine return to approximately but not exactly the same position each time; introducing "dither" to avoid unwanted synchronism with external processes; and correcting for nonuniformity in some process downstream from the random number generator.

Traditionally, nonuniform distributions are generated by transforming the output of a uniform random-number generator, often using elaborate floating-point computations. In this paper I outline an alternative that is especially suitable for the latter set of cases, where quick computation is more important than hitting a specified distribution exactly.[1]

2. Approach

As Fig. 1 shows, the sum of n uniform random variables is an $(n-1)$th-degree polynomial approximation to a normal distribution (bell curve) ([2], p. 22). For practical purposes, the curve with $n=3$ is smooth enough.

Thus, one can generate a bell curve distribution by merely generating three random numbers each time, adding them, and dividing by three.

Bell curves can be used like wavelets to synthesize more complex curves. For example, the distribution in Fig. 2 was synthesized from a nonzero baseline mixed with three bell curves by the code in Fig. 5.
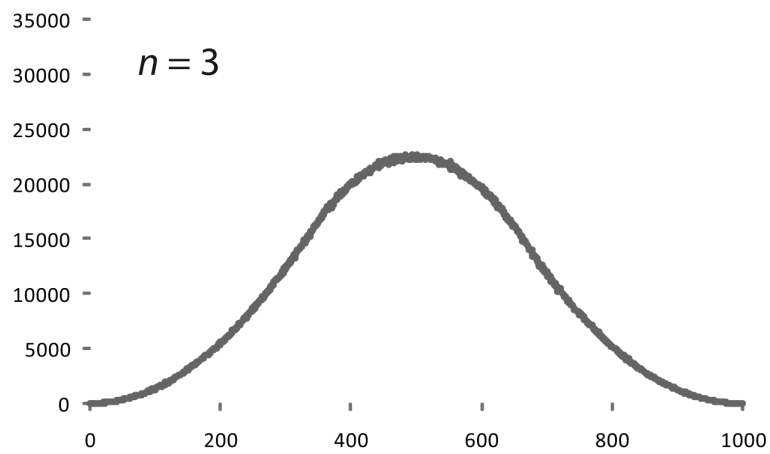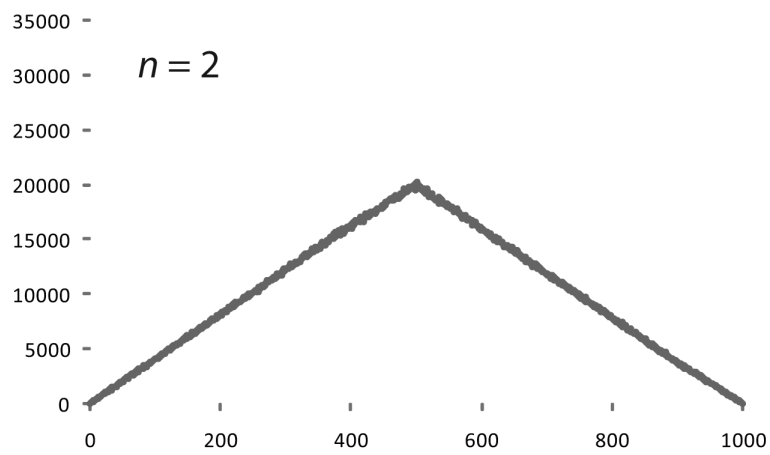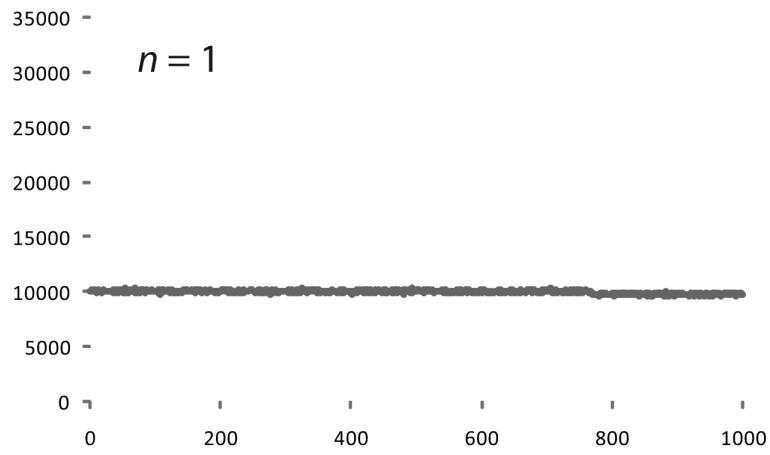
Figure 1: Histograms of the sum of n uniform random variables, from $10\,000\,000$ trials.
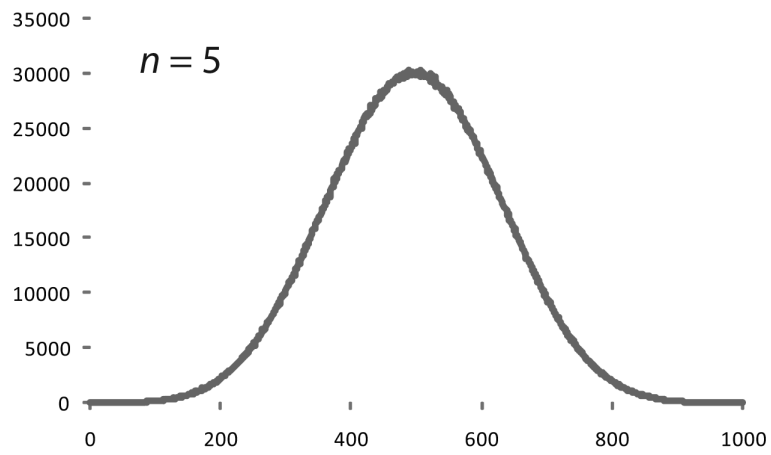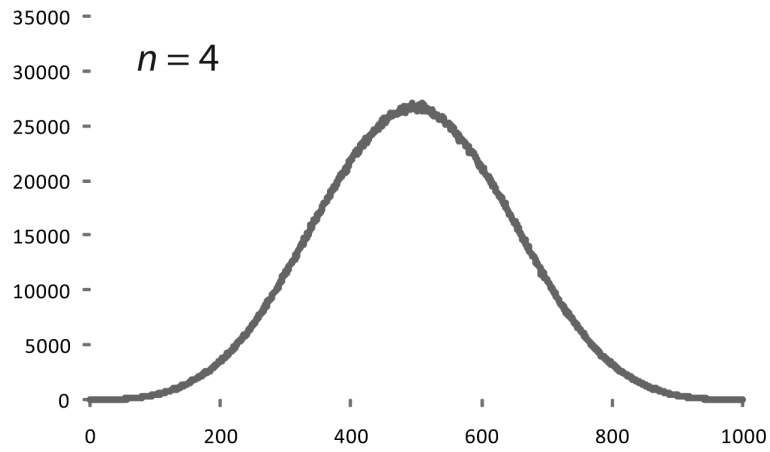
$n = 4$
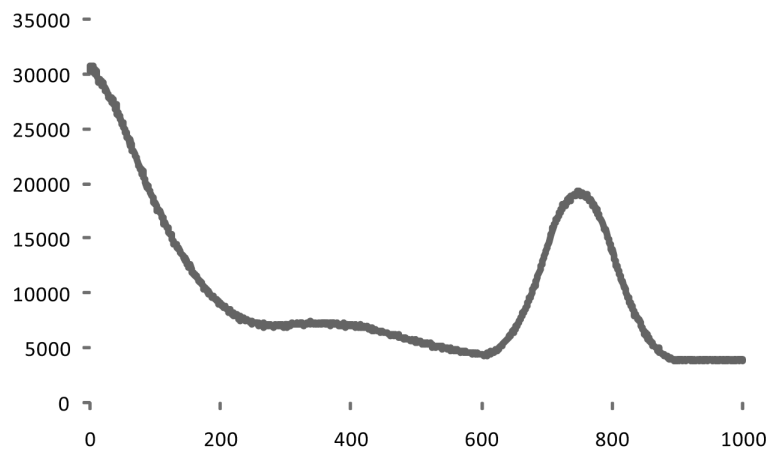
$n = 5$

Fig. 1, continued.

Figure 2: Histogram of custom random number generator in Fig. 5.

```
int genrand(int bmin, int bmax, int rmin, int rmax, int n) {
    // Generalized random number generator;
    // sum of n random variables (usually 3).
    // Bell curve spans bmin<=x<bmax; then,
    // values outside rmin<=x<rmax are rejected.
    int i, u, sum;
    do {
        sum = 0;
        for (i=0; i<n; i++) sum += bmin + (rand() % (bmax - bmin));
        if (sum < 0) sum -= n-1;            /* prevent pileup at 0 */
        u = sum / n;
    } while ( ! (rmin <= u && u < rmax) );
    return u;
}
```

Figure 3: Generalized random number generator (sum of n uniform random variables).

Bell curves lack one property of wavelets [3]: they do not have an average value of zero, and in fact they never dip below zero at all. Thus, adding another bell curve to a synthesized function can only raise it, not lower it. For histograms, this is not a serious objection because the height of the curve has only relative significance; the whole histogram can be raised or lowered by generating more or fewer random numbers.

To understand that bell curves can be used like wavelets, consider some arbitrary distribution $f(x)$, some approximation to it $g(x)$, and the difference $g(x) - f(x)$. Either the difference is a constant, in which case it can be filled by mixing in a uniform distribution, or else it has one or more "humps." Fill one of the humps with a bell curve sized to fit it, and you have a better $g(x)$ and a new $g(x) - f(x)$ from which that hump is missing. Repeat as desired until the fit is sufficiently close.

3. Implementation and testing

To put this idea to the test, the all-integer algorithm shown in Fig. 3 was coded in C and executed under the Plan 9 operating system [4] on a 1.87-GHz Pentium processor. Each call to *genrand* with n=3 took, on the average, less than 0.01 microsecond. The histograms in all the illustrations were made with this function, by downloading its output to a PC and graphing with Microsoft Excel.

The arguments of the function specify two ranges, the range that the bell curve should span and the range of values acceptable as output. Thus, it is possible to compress or truncate the histogram (Fig. 4). Naturally, if a substantial part of the bell curve is discarded, CPU time is wasted, but this is still a quick way to generate a partial bell curve.

In the algorithm, if the sum of individual random variables is negative, it is decremented by $n-1$ before performing the integer division by n. The reason is that without this step, there are more ways to get 0 than any other number. For example, if n=3, then not only do 2/3, 1/3, and 0 truncate to 0, so do $-1/3$ and $-2/3$. By shifting all negative results farther negative by $-2/3$, we get the latter two to truncate to -1.

Fig. 5 shows how the synthesis of a distribution is done. This particular function has a 40% chance of choosing the first call to *genrand,* a 30% chance of choosing the second, a 20% chance of choosing the third, and a 10% chance of choosing the fourth. Thus, the bell curves are mixed in the desired proportions.

One limitation inherent in this technique is that the random numbers need to fall within in
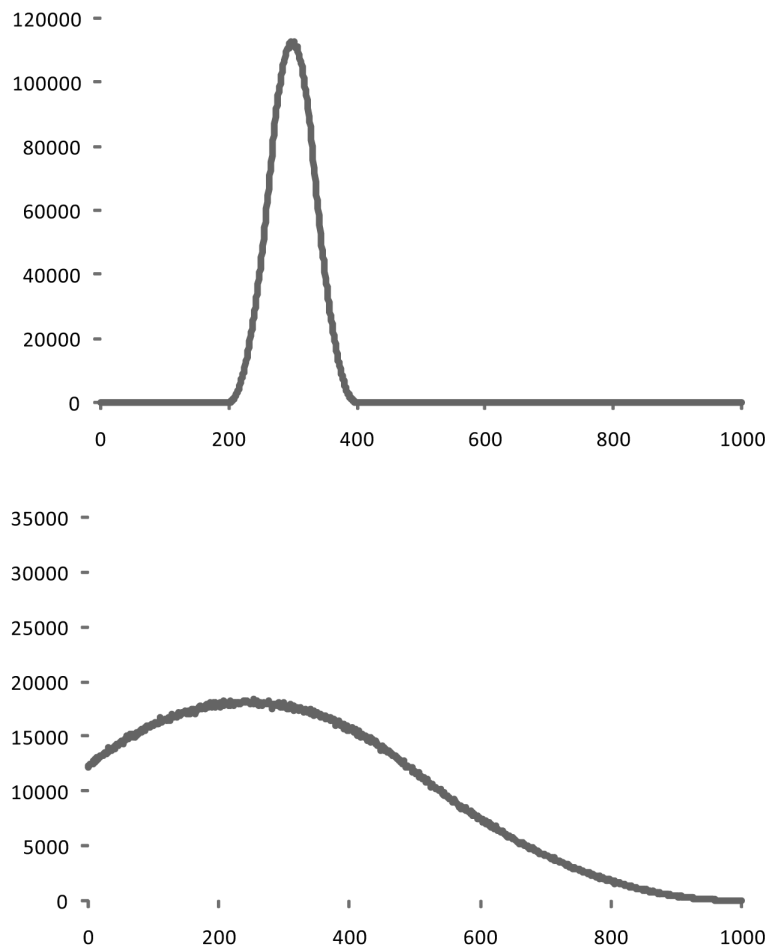
Figure 4: Generated third-degree ($n = 3$) bell curve spans any specified range of values. Bell need not fit within desired range; in that case, values outside range are generated and rejected.

```
int customrand(void) {
    switch (rand() % 10) {
        case 0:
        case 1:
        case 2:
        case 3:
            return genrand(0,1000,0,1000,1);    // flat baseline
        case 4:
        case 5:
        case 6:
            return genrand(-400,300,0,300,3);   // large peak beyond left edge
        case 7:
        case 8:
            return genrand(600,900,600,900,3);  // peak at 750
        default:
            return genrand(0,700,0,700,3);      // very low, broad peak at 350
    }
}
```

Figure 5: Sample code to interleave calls to genrand with different parameters, to combine multiple bell curves into the single distribution shown in Fig. 2.

a range considerably smaller than that of the underlying built-in random number generator. In Plan 9 C, *rand* produces integers from 0 to 32 767 inclusive. When taken modulo 1000, as in the examples, these are not quite uniform because (for example) there are 32 ways to get 767 but only 31 ways to get 768. This nonuniformity is just visible in the topmost curve in Fig. 1 but is usually negligible. If the modulus were 10 000 or 20 000, it would be serious.

References

[1] P. Bratley, B. L. Fox, and L. E. Schrage, *A Guide to Simulation,* 2nd edn. New York: Springer-Verlag, 1987.

[2] L. Devroye, *Non-Uniform Random Variate Generation.* New York: Springer-Verlag, 1986.

[3] I. Daubechies, "Wavelet transforms and orthonormal wavelet bases," in *Different Perspectives on Wavelets,* I. Daubechies, Ed. (Proceedings of Symposia in Applied Mathematics, 47.) Providence, R.I.: American Mathematical Society, 1993, pp. 1–33.

[4] Thompson, Ken (1990) "Plan 9 C compilers."
http://plan9.bell-labs.com/sys/doc/compiler.pdf