# Generating Pseudo-random Floating-Point Values

Allen B. Downey

July 25, 2007

**Abstract**

The conventional method for generating a pseudorandom floating-point value is to generate a pseudorandom integer and divide by a constant (using floating-point arithmetic). The problems with this approach are (1) the possible outcomes are a small subset of the representable floating-point values (about 7% for a typical implementation) and (2) subsequent transformation of these values may not yield the expected distributions (for example, applying a log transformation to numbers from a uniform distribution does not yield an exponential distribution as it would in real arithmetic). We present a new algorithm with the following properties: (1) it can produce every representable floating-point value in a given range, and (2) it is efficient in both time and use of pseudorandom bits.

## 1 Introduction

There has been much work on the problem of generating pseudo-random integers with statistical properties appropriate for such applications as Monte Carlo simulation and stochastic modelling. In general, though, this work has given little attention to the problem of generating random floating-point values.

(I would like to write a section here explaining the impact of my algorithm on some real applications, assuming it has one – in the meantime, I don't have much to say, other than to explain the algorithm.)

## 2 Conventional methods

The usual way to generate random floating-point values uniformly-distributed in the range $[0.0, 1.0]$ is to generate a random integer in the range $[0, m]$, convert the integer to floating-point, and then divide by $m - 1$ using floating-point arithmetic.

To illustrate the problems with this approach, we will consider the following example.

```
#define RAND_MAX 2147483647.0     /* = 2^31 - 1  */
int x = random ();
float f = (float) x / RAND_MAX;
```

where `random()` returns a random integer in the range $[0, 2^{31} - 1]$, the type `int` indicates a 32-bit, 2's complement number and the type `float` indicates an IEEE Standard single-precision floating-point number.

There are several problems with this approach, the first of which is a matter of implementation, but the latter of which are fundamental algorithmic flaws.

The implementation issue is that the number $2^{31} - 1$ is too large to be represented exactly in single-precision. The nearest representable values are $2^{31} = 2147483648$ above and 2147483520 below. Since the denominator is rounded up, the mean of this distribution will be very slightly too low.

But this is not a serious problem; the error is less than 5 parts in 10 trillion, which is acceptable for most applications. Furthermore, this error can be eliminated by performing the division in double-precision arithmetic, as in:

```
#define RAND_MAX 2147483647.0
int x = random ();
float f = (double) x / RAND_MAX;
```

The algorithmic flaw is more serious and harder to fix. Generating random floating-point values by division makes it impossible to produce the vast majority of floating-point values in the range. Of the $2^{30}$ representable floating-point values between 0.0 and 1.0, this method can produce only about $2^{26}$. This estimate is based on the number of representable floating point values in the range $[0, 2^{31}]$. The integers from 1 to $2^{24}$ can be represented exactly. In each binade from $2^{24}$ to $2^{31}$, there are $2^{23}$ representable values. So the total is $2^{24} + 7 \cdot 2^{23}$ which is $2^{26.17}$. All representable values in the range can be generated by this algorithm. So about 7% of the values in the range can occur.

Figure 1 illustrates a mapping between equally-spaced real numbers and the representable floating-point values. The floating-point values are not equally-spaced on the number line. In the range of small values, the floating-point numbers are much denser than the real values we are trying to represent; thus, only a few of the floating-point values will ever be selected. In the range of large values, the floating-point numbers are much sparser than the real values; thus, many real values will be rounded to the same floating-point value.

## 2.1 Transformations

One common way to generate exponentially-distributed numbers is to choose numbers from a random distribution and apply a logarithmic transformation [Press et al.]. Example:

```
float f = (double) random () / RAND_MAX;
float e = -log (f);
```

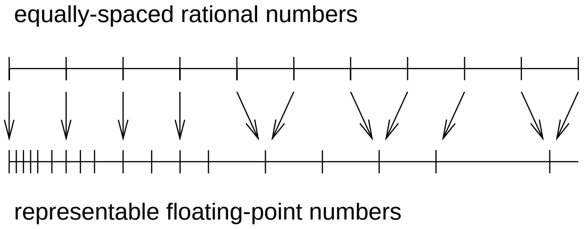


Figure 1: Mapping equally-spaced real numbers onto the representable floating-point values.

One problem with this approach is that since $f$ may be equal to zero, the call to $log()$ may result in a domain error. The more important problem is that since the smallest value for $f$ is $\sim 2^{-31}$, the largest value for $e$ is $\sim 9.3$. Of course, a real exponential distribution can produce arbitrarily large values.

The algorithm presented in this paper can produce the smallest representable floating-point value, which is $\sim 2^{-149}$ in single precision. Thus, if the logarithmic transformation is applied to these values, the result is an exponential distribution truncated at $\sim 44.9$.

Although this is an improvement, it still does not satisfy the goal of producing any representable value in the range $[0, \infty]$.

(I have given some thought to fixing this problem, but so far I have hit a wall).

# 3 Proposed algorithm

The goal of this algorithm is to generate uniformly-distributed pseudo-random floating-point values in the range $[0, 1]$, assuming that we are given an algorithm for generating pseudo-random bits. These bits should have probability 50% of being in each of two states, and be statistically independent.

Our approach is to choose floating-point values in the range such that the probability that a given value is chosen is proportional to the distance between it and its two neighbors. The algorithm works in two steps, first choosing the exponent range of the value, then choosing the mantissa.

We will use a simple example to demonstrate the principle, then show an implementation for IEEE Standard single-precision floating-point values.

## 3.1 Example

Figure 2 shows the set of representable values for a small floating-point representation:

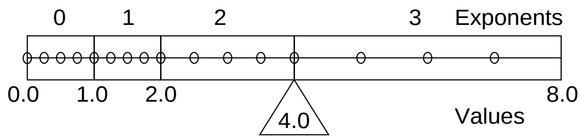


Figure 2: The representable values for a small floating-point representation.

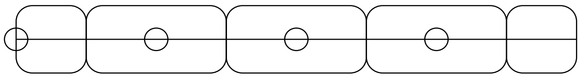

Figure 3: Representable values in an exponent range, and their probability of being chosen.

$$x = 1.dd \times 2^{exp}$$

where $dd$ represents a two-digit mantissa and $exp$ can be one of 1, 2, 3. The exponent $exp = 0$ is special, indicating a subnormal value. These values are interpreted as:

$$x = 0.dd \times 2^{exp}$$

The key observation here is that the exponent ranges shown in Figure 2 balance on the center-point 4.0 (marked with a triangular fulcrum). Thus if a random value were chosen in the range $[0, 8]$ the probability that the value will be in the highest exponent range is 50%. This property is true for any range of floating-point values with radix 2 (although a small adjustment is required in the absence of subnormals).

Thus, the algorithm for choosing an exponent range is:

1. Choose a single random bit.
2. If it is 0, choose the largest exponent (the area to the right of the fulcrum).
3. Otherwise, choose an exponent from the ranges to the left of the fulcrum.

The number of random bits generated may be as large as the difference between the largest and smallest exponent in the range, but on average the number of bits is less than 2.

Having chosen an exponent range, we now have to choose among the representable values within that range. Again, the goal is to choose each value with probability proportional to the distance between it and its two neighbors. Choosing two random bits *almost* satisfies this goal; with a small adjustment it can be made correct.

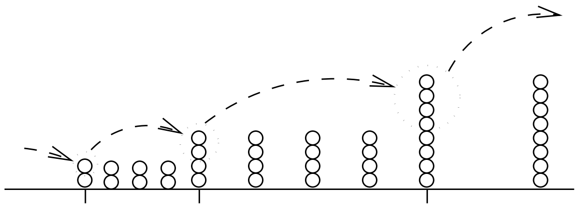

Figure 4: Probability of selecting a given floating-point value, before the shift.

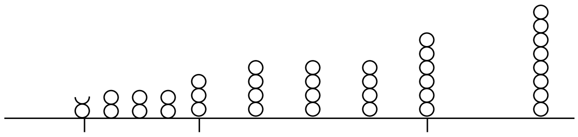

Figure 5: The corrected probability for each value is proportional to the distance between the value and its two neighbors.

Figure 3 shows the four representable values in an exponent range. The rounded rectangles indicate the area associated with each value — the probability of choosing a particular value is proportional to the area of its rectangle.

Choosing a random two-bit mantissa would give equal probability to all values in the exponent range. This is almost correct, except for the first point in the range, which should be chosen half as often, and the first point in the next exponent range, which will never be chosen, but which should be chosen as often as the first.

The simple solution, then, is:

1. Choose a random mantissa.
2. If it is 0, then with probability 50% increase the exponent by one.

Figure 4 shows a set of values spanning several exponent ranges. The height of each column indicates the probability that a given value will be chosen. The initial column heights indicate the probabilities that would result without Step 2, above. The arrows indicate the shift in probability that results from Step 2.

Figure 5 shows the outcome of this shift. As desired, the probability that each value is chosen is proportional to the distance between it and its two neighbors.

# 4 Implementation

The following code shows an implementation of the algorithm presented in the previous section. It is intended to be clear, not efficient. Obvious optimizations include inlining `get_bit()` and using the 8 random bits that are leftover each time we generate a random mantissa.

```
/* BOX: this union is used to access the bits
of floating-point values */

typedef union box {
  float f;
  int i;
} Box;

/* GET_BIT: returns a random bit.  For efficiency,
   bits are generated 31 at a time using the
   C library function random () */

int get_bit ()
{
  int bit;
  static bits = 0;
  static x;

  if (bits == 0) {
    x = random();
    bits = 31;
  }
  bit = x & 1;
  x = x >> 1;
  bits--;
  return bit;
}

/* RANDF: returns a random floating-point
   number in the range (0, 1),
   including 0.0, subnormals, and 1.0 */

float randf ()
{
  int x;
  int mant, exp, high_exp, low_exp;
  Box low, h, ans;

  low.f = 0.0;
  high.f = 1.0;
```

```
  /* extract the exponent fields from low and high */
  low_exp = (low.i >> 23) & 0xFF;
  high_exp = (high.i >> 23) & 0xFF;

  /* choose random bits and decrement exp until a 1 appears.
     the reason for subracting one from high_exp is left
     as an exercise for the reader */

  for (exp = high_exp-1; exp > low_exp; exp--) {
    if (get_bit()) break;
  }

  /* choose a random 23-bit mantissa */
  mant = random() & 0x7FFFFF;

  /* if the mantissa is zero, half the time we should move
     to the next exponent range */
  if (mant == 0 && get_bit()) exp++;

  /* combine the exponent and the mantissa */
  ans.i = (exp << 23) | mant;

  return ans.f;
}
```

# 5 Performance

We compare the performance of this algorithm with the standard implementation presented above:

```
#define RAND_MAX 2147483647.0
int x = random ();
float f = (double) x / RAND_MAX;
```

On my SPARCStation 2, the standard algorithm takes $\sim 332\mu s$, including the procedure call overhead. The proposed algorithm takes, on average, $\sim 327\mu s$, also including overhead. The particular values of these numbers are not important, since they depend on details of the architecture and implementation; the point is that the proposed algorithm can be implemented as efficiently as the standard algorithm, and at the same time offer the desireable properties described above.

The conventional approach generates 31 random bits, but can generate only $\sim 2^{26}$ different values. Thus, in some sense it wastes 5 bits. The proposed algorithm uses, on average, 25 random bits, and generates $2^{30}$ different values.

(Which by the same obviously fallacious logic implies that it is summoning 5 bits of randomness from nowhere.)

## 6 Conclusions

(I think this algorithm is a good thing, but I need to show that the problems I have identified affect real applications, and that my algorithm fixes them, or at least provides a way to do the same thing more efficiently.)