HAL open science

Drawing random floating-point numbers from an interval

Frédéric Goualard

▶ To cite this version:

Frédéric Goualard. Drawing random floating-point numbers from an interval. 2021. hal-03282794v1

HAL Id: hal-03282794

https://hal.science/hal-03282794v1

Preprint submitted on 9 Jul 2021 (v1), last revised 7 Oct 2022 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A

Drawing random floating-point numbers from an interval

FRÉDÉRIC GOUALARD, CNRS, LS2N, UMR 6004

Drawing a floating-point number uniformly at random from an interval $[a, b)$ is usually performed by a location-scale transformation of some floating-point number drawn uniformly from $[0, 1)$. Due to the weak properties of floating-point arithmetic, such a transformation cannot ensure respect of the bounds, uniformity or spatial equidistributivity. We investigate and quantify precisely these shortcomings while reviewing the actual implementations of the method in major programming languages and libraries, and we propose a simple algorithm that avoids these shortcomings without compromising performances.

Categories and Subject Descriptors: G.1.0 [**General**]: Computer arithmetic; Error analysis

General Terms: Algorithms, Reliability, Experimentation

Additional Key Words and Phrases: floating-point number, IEEE 754 standard, random float

1. INTRODUCTION

To simulate the throw of a fair six-sided die, most people would know better than to take the modulo of some large random positive integer: as pointed out by Lemire [Lemire 2019], such a method favors the less random bits of weak Random Number Generators (RNGs); it destroys even so slightly the uniformity of the process whenever 6 is not a divisor of the number of different outcomes offered by the RNG; and lastly, it can be quite expensive as it relies on division. Whatever the knowledge and skills of a software developer, major programming languages, or their standard libraries, usually offer functions to directly draw integers from an interval whose implementation is both efficient and correct regarding the uniformity of the outcome.

When random floating-point numbers ("*floats*", for short) are needed, the functions available in most programming languages start by computing a random float x in $[0, 1)$ and apply the affine transformation $y = a + (b - a)x$ to get a value y in the interval $[a, b)$, as would be done with real numbers. However, the properties of floats as defined by the ubiquitous IEEE 754 standard [IEEE 2019] lead to several problems, among which are a lack of uniformity of the outcome and the generation of values outside the requested domain.

In Section 2.1, we present just enough of IEEE 754 binary floating-point arithmetic necessary to understand the consequences of its use on the generation of random floats (Section 2.2). Readers already familiar with the IEEE 754 standard may skip Section 2.1 and only refer to Table I for the notations used.

We review in Section 3 the documentation and the implementation of fifteen programming languages and libraries regarding the generation of floating-point numbers in an interval, pointing out the shortcomings and/or errors in both the documentation and the code.

In Section 4, we perform a detailed error analysis of the methods presented in Section 3.

Lastly, we propose in Section 5 a simple algorithm to compute random floats in an interval, which avoids the problems plaguing the methods reviewed in Section 3, and we compare its performances with theirs.

Author's address: F. Goualard, Université de Nantes. CNRS, LS2N, UMR 6004, 2 rue de la Houssinière, BP 92208, F-44322 NANTES CEDEX 3.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© YYYY Copyright held by the owner/author(s). 1049-3301/YYYY/01-ARTA $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

2. ON FLOATING-POINT ARITHMETIC AND ITS CONSEQUENCES

Computation over the reals is usually approximated by using binary floating-point arithmetic as specified by the IEEE 754 standard [IEEE 2019]. This standard is supported by the vast majority of computers nowadays, even though some compatibility may be withheld from the user at the software level (Python, for example, raises an exception whenever a division by zero occurs, while the sanctioned IEEE 754 behavior is to return an infinite value when the dividend is not zero). Though the properties of the reals do not all translate to the floats, many algorithms are implemented as if it were the case.

The following section presents the bare minimum to understand the rest of this article. The books by Higham [Higham 2002] and Muller *et al.* [Muller 2010] are good references, should the reader long for more detailed and less fast-paced presentations.

2.1. IEEE 754 floating-point numbers

Binary floating-point numbers complying with the IEEE 754 standard exist in different formats with varying precisions and ranges. The format of a set of floats $\mathbb{F}_p^{\text{emax}}$ can be completely defined by the pair of integers (p, emax); in the rest of this article, we assume $p \geqslant 2$. A float $x \in \mathbb{F}_p^{\text{emax}}$ is made of a *significand* —which is a fractional number m in the interval $[0, 2 - 2^{1-p}]$—, a sign s, and a scale factor 2^E such that $x = (-1)^s \times m \times 2^E$. The domain for E is $[1 - \text{emax}, \text{emax}]$. Figure 1 shows all the floats from $\mathbb{F}_3^1$ on the real line. Given a float x, $\underline{\gamma}(x)$ is the distance between x and its predecessor $\mathsf{prev}\,(x)$ on the real line; $\overline{\gamma}(x)$ is the distance between x and its successor $\mathsf{next}\,(x)$. The two distances are the same except when x is a power of two (written $\mathsf{P}_2(x)$). The set of floats in $\mathbb{F}_p^{\text{emax}}$ contains two "infinities": the operations on $\mathbb{F}_p^{\text{emax}}$ are defined in such a way that when a result is too large to be represented by finite floats (*overflow*), it is replaced by the correctly signed infinity.

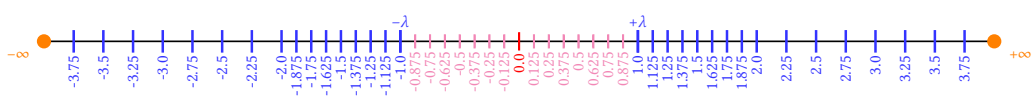


Fig. 1: The real line with $\mathbb{F}_3^1$ floats. Blue values are normal floats; magenta values are subnormal floats. The value λ is the smallest positive normal float.

Floats with a significand m in the interval $[1, 2 - 2^{1-p}]$ are *normal floats*. The smallest normal float is named $\lambda(\mathbb{F}_p^{\text{emax}})$[1], and the largest, $\mathsf{floatmax}(\mathbb{F}_p^{\text{emax}})$. The floats with a significand in $[0, 1 - 2^{1-p}]$ are *subnormal floats*. Subnormal floats all have the same scale factor equal to $2^{1-\text{emax}}$. They represent very small floats. An *underflow* happens when a number smaller than λ occurs in a computation.

Floats from a format $\mathbb{F}_p^{\text{emax}}$ are represented in memory by three fields: the sign bit s, the exponent E (stored in memory as the *biased* positive value $e = E + \text{emax}$) and the *fractional part* f of m (that is, the $p - 1$ bits to the right of the fractional point). Figure 2 shows the representation of the binary32 single precision format and the binary64 double precision format, which are two of the formats explicitly specified by the IEEE 754 standard.

When the result of a computation involving floats is not itself a float, *rounding* takes place. The IEEE 754 standard imposes that all arithmetic operations be *correctly rounded*: the result must be computed as if infinite precision was available, and then approximated by the closest float. For example, considering $\mathbb{F}_3^1$, adding 1.0 to 1.875 gives 2.875, which is

[1]Or simply λ, when the set of floats is implicit.

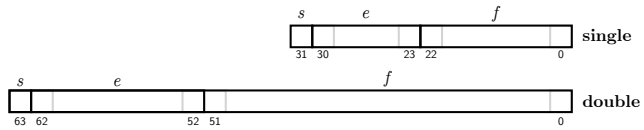


Fig. 2: Representation of binary32 and binary64 IEEE 754 floating-point numbers in memory.

not representable. That value is therefore rounded to the closest float. The two surrounding floats are 2.75 and 3.0, which are exactly at the same distance. The rule in that case is to round to the float x whose internal representation (Figure 2) has its rightmost bit equal to zero (noted $\mathsf{fraceven}(x)$), 3.0 here.

Given some set of floats, the correctly rounded value of a real $r \in \mathbb{R}$ will be written $\mathsf{fl}(r)$. The rounded result of a computation C will be written $\mathsf{fl}\langle C\rangle$ (note the angled brackets replacing the parentheses to indicate that rounding takes place recursively at all nodes of the computation, not solely at the highest level).

Example 2.1. Given $(a, b, c) \in \mathbb{F}_p^{\mathrm{emax}}$, we have:

$$\mathsf{fl}\langle a + bc\rangle = \mathsf{fl}(a + \mathsf{fl}(bc))$$

A property that will be used repeatedly in this article is the *monotonicity of rounding*:

$$\forall (r_1, r_2) \in \mathbb{R}^2 \colon r_1 \leqslant r_2 \implies \mathsf{fl}(r_1) \leqslant \mathsf{fl}(r_2)$$

Requiring a correct rounding of all arithmetic operations brings important benefits in that it allows to compute the maximum error for a computation involving more than one operation. In particular, given an operator $\top \in \{+, -, \times, \div\}$, we will use the following:

$$\forall r \in \mathbb{R} \colon \mathsf{fl}(r) = r(1 + \delta), \text{ with } \delta \in [-2^{-p}, 2^{-p}]$$

and

$$\forall (x, y) \in \mathbb{F}_p^{\mathrm{emax}} \times \mathbb{F}_p^{\mathrm{emax}} \colon \mathsf{fl}(x \top y) = (x \top y)(1 + \delta), \text{ with } \delta \in [-2^{-p}, 2^{-p}]$$

provided no underflow occurs. In case of underflow, other rules apply but we will not present them here as we will always assume the absence of underflow.

Table I summarizes all the notations introduced in this section that will be used in the rest of the article.

2.2. Drawing floats uniformly at random

Even though it might be more useful to generate floating-point numbers in the open interval $(0, 1)$, it is often more convenient, from an implementation standpoint, to consider the half-closed interval $[0, 1)$ and use a rejection procedure to get rid of unwanted values, if necessary.

Consider the set of floats $\mathbb{F}_3^3 \cap [0, 1]$ illustrated in Figure 3 and let us note $\mathbb{S}_0^1$ the set of floats that can be drawn at random from $[0, 1)$. Since the distance between a float and the next doubles for every power of two, a procedure to draw floats uniformly at random cannot simply return any float from the interval $[0, 1)$. We easily see that the distance between 1.0 and $\mathsf{prev}\,(1.0)$ directly dictates the maximum number of floats we can sample uniformly in $[0, 1)$: if we draw values in the set $\mathbb{S}_0^1 = \{k2^{-p} \mid 0 \leqslant k < 2^p\}$, they are all perfectly representable uniformly-spaced floats in $[0, 1)$; considering a set with less than 2^p values is wasteful (we could draw more); on the other hand, allowing more than 2^p values destroys uniformity.

Some authors (e.g., Marsaglia and Tang [Marsaglia and Tsang 2004], or Holian *et al.* [Holian et al. 1994]) have proposed algorithms to generate random floating-point numbers directly. However, most programming languages and libraries surveyed seem to resort to one of the two following procedures:

Table I: Notations introduced in Section 2.1.

Notation	Meaning
$\mathbb{F}_p^{\text{emax}}$	Set of floats with p bits significand and $\log_2(\text{emax}+1)+1$ bits exponent
$\mathsf{fraceven}(x)$	True if the rightmost bit of the fractional part of x is 0
$\mathsf{fl}(r)$	Real r rounded to the nearest-even float
$\mathsf{fl}\langle e\rangle$	Floating-point evaluation with the expression e rounded to nearest-even
$\mathsf{floatmax}(T)$	Largest finite float of type T
$\underline{\gamma}(x)$	Distance between the float x and $\mathsf{prev}\,(x)$
$\overline{\gamma}(x)$	Distance between the float x and $\mathsf{next}\,(x)$
$\gamma(x,y)$	$\max(\overline{\gamma}(x),\underline{\gamma}(y))$, for $x<y$
λ, $\lambda(\mathbb{F}_p^{\text{emax}})$	Smallest positive normal float in $\mathbb{F}_p^{\text{emax}\dagger}$
$\mathsf{next}\,(x)$	Float to the immediate right of the float x on the real line
p	Number of bits in the significand of a float
$\mathsf{P}_2(x)$	True if the float x is a power of two
$\mathsf{prev}\,(x)$	Float to the immediate left of the float x on the real line

† The indication of the set of floats is dropped when it can be deduced from the context.

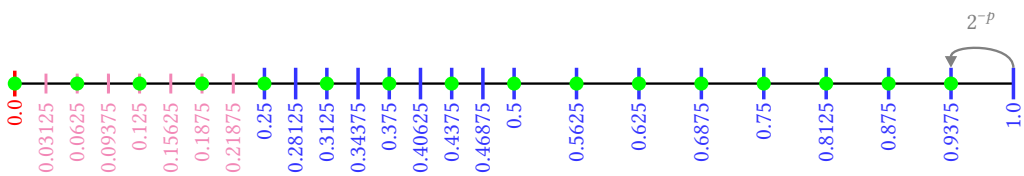


Fig. 3: Set of floats from $\mathbb{F}_3^3$ in $[0,1]$. Green dots identify the elements of the largest set of spatially equidistributed floats in $[0,1)$.

— Generate a random integer in the interval $[0,2^k-1]$ and divide it by 2^k to obtain a float in $[0,1)$. As seen above, the best choice for k is p but some languages use other values [Goualard 2020];
— Generate a fractional part as a $p-1$ bits random integer, add the integer "emax" shifted to the left by $p-1$ positions, and reinterpret the resulting integer as the IEEE 754 representation of a floating-point number in the domain $[1,2)$. It then suffices to subtract 1.0 from it to obtain a float in $[0,1)$.

The second procedure can only generate 2^{p-1} different floats, differing all by their $p-1$ bits fractional part. We then have: $\mathbb{S}_0^1 = \{k2^{1-p} \mid 0 \leqslant k < 2^{p-1}\}$. That is only half the number of floats that could legitimately be drawn from $[0,1)$.

Once a float x has been drawn from $[0,1)$, it is affinely transformed into a float y in the interval $[a,b)$ by the formula:

$$y = \mathsf{fl}\langle a+(b-a)x\rangle \tag{1}$$

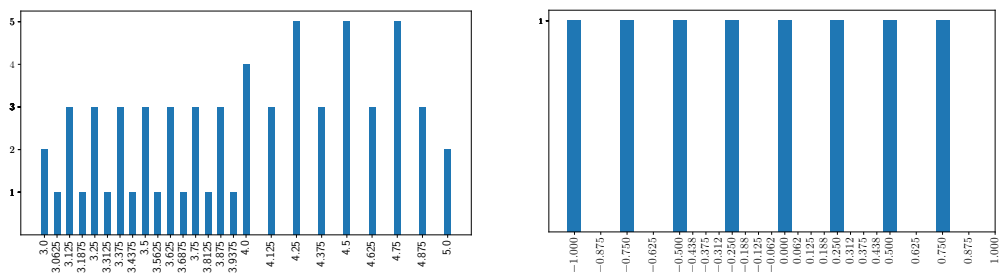
That transformation poses several problems that are presented without any theoretical analysis in the following paragraphs. We will quantify them precisely from a theoretical standpoint in Section 4.

The quantity $b - a$ may overflow if b and a have opposite signs and are large in absolute value. This is often not taken care of by actual implementations, as we will see in Section 3. Consider, for example, the following commands in Julia:

```
julia> using Random, Distributions

julia> rand(Uniform(-1.8f38,1.8f38))
Inf
```

Trying to draw a binary32 single precision float in the domain $[-1.8 \cdot 10^{38}, 1.8 \cdot 10^{38}]$, we always get an infinite value since $1.8 \cdot 10^{38} - -1.8 \cdot 10^{38}$ overflows.[2]



(a) Number of occurrences of each float in $[3, 5)$ using an affine transformation of all floats from $\mathbb{S}_0^1$ with $\mathbb{F}_5^{15}$.

(b) Number of occurrences of each float in $[-1, 1)$ using an affine transformation of all floats from $\mathbb{S}_0^1$ with $\mathbb{F}_2^3$.

Fig. 4: Some examples of problems induced by computing an affine transformation illustrated with small float sets.

By the *pigeonhole principle*, if $[a, b)$ contains less than 2^p floats, the affine transformation will map —through rounding errors— several floats from $[0, 1)$ to the same float in $[a, b)$, thereby potentially destroying uniformity. This situation happens very easily, since the distance between adjacent floats increases every power of two. Figure 4a shows an example with $\mathbb{F}_5^{15}$. The histogram presents the number of floats from $\mathbb{S}_0^1$ that are mapped to each float in $[3, 5)$. Note that if each float in $\mathbb{S}_0^1$ has the same probability to be drawn, the probability of each float in $[3, 5)$ to be drawn will not be uniform, as some floats are mapped to five times more than others. In addition, due to rounding errors, the value 5.0 is among the values drawn, while we were expecting values from $[3, 5)$.

Rounding errors may also induce a lack of uniformity, even when $[a, b)$ contains a number of floats that is equal or greater to the one in $\mathbb{S}_0^1$, even more so when the distance between some adjacent floats in $[a, b)$ is smaller than 2^{-p}. Consider, for example the following Julia program:

[2]Note that, even though the internal computation in `Uniform` seems to use single precision floats, the result is the double precision infinite value `Inf` instead of the single precision `Inf32`.

```
p = 24
a = 0.25f0; b = 1.0f0
L = [a+(b-a)*x for x in 0.0f0:2.0f0^-p:(1.0f0-2.0f0^-p)]
Lcount = counter(L)
println("Variance of distances between adjacent values in L: ",
        var(diff(sort(collect(keys(Lcount))))))
println("Number of different floats in L: $(length(Lcount))")
println("Number of floats in [a,b): $(nb_fp_numbers(a,b)-1)")
```

The vector `L` contains the affine transformation of all values in $\mathbb{S}_0^1$ for the binary32 single precision format; `Lcount` is a dictionary associating for each unique value in `L` its number of occurrences; the function `nb_fp_numbers()` computes the number of floats between its two arguments, themselves included. The output of the code is:

```
Variance of distances between adjacent values in L: 1.4210864e-16
Number of different floats in L: 13981013
Number of floats in [a,b): 16777216
```

There are exactly 2^{24} values in $[0.25, 1) \cap \mathbb{F}_{24}^{127}$ but only 13981013 of them ($\approx 83\,\%$) are mapped to. Hence, some values in L occur necessarily more often than others; in addition, the variance of the distances between adjacent values mapped to in $[0.25, 1)$ is non-null, showing that the values that can be drawn in $[0.25, 1)$ are not spatially equidistributed.

When $[a, b)$ contains more values than $\mathbb{S}_0^1$, it is wasteful to only map the 2^p values from $\mathbb{S}_0^1$. Consider, for example the domain $[-1, 1]$. There are 2^{p+1} equidistant values that could be drawn from the domain. By mapping only 2^p values, we consciously waste half of them (see Figure 4b for an example in $\mathbb{F}_2^3$).

Lastly, up to now we have only considered the problem of drawing values in an half-closed interval $[a, b)$, which is dictated by the fact that most languages and libraries first draw floats in $[0, 1)$. We have already encountered one problem, viz. that through rounding errors, we may nevertheless draw the right bound b. We will see in Section 4 some sufficient conditions for this event not to happen. Rounding errors hinder the devising of procedures that would draw floats from intervals with specific bounds (left-open/right-closed, ...) with any certainty.

3. DRAWING FLOATS FROM AN INTERVAL: HOW DO THEY DO IT?

All major programming languages provide functions, directly or through standard libraries, to draw floating-point numbers uniformly at random from the intervals $[0, 1)$ and/or $(0, 1)$. We review in this section the facilities offered by fifteen programming languages to draw floats uniformly from a more general interval $[a, b)$ (or, sometimes, (a, b)); we also evaluate the corresponding implementation, either as mandated by the standard of the language, or the way it is actually programmed in popular compilers/interpreters.

3.1. C

The C language does not have any function to draw a float from an interval $[a, b)$. On the other hand, the GNU Scientific Library, which might be considered the closest thing to a C standard numerical library, offers the `gsl_ran_flat()` function. According to its documentation, given a random float $x \in [0, 1)$, the function computes a float $y \in [a, b)$ with the formula:

$$y = a(1 - x) + bx \tag{2}$$

Over the reals, Equation. (2) is equivalent to $y = a + (b - a)x$. Over the floating-point numbers, it has, however, the advantage of avoiding an overflow when a and b are large and

of opposite signs. Unfortunately, as the example below shows, Formula (2) does not protect against drawing the right bound:

```
int main(void)
{
    gsl_rng *rng = gsl_rng_alloc(gsl_rng_mt19937);
    gsl_rng_set(rng,43);
    const double a = 3.5;
    const double b = 3.5000000004656613;
    double y;
    unsigned int i = 0;
    do {
      ++i;
      y = gsl_ran_flat(rng,a,b);
    } while (y != b);
    printf("Right bound obtained after %u iterations!\n",i);
    gsl_rng_free(rng);
}
```

The output of the program is:

```
Right bound obtained after 2751203 iterations!
```

As we will see in Section 4.3, Formula 2 is also not numerically stable and its evaluation may incur very large rounding errors, thereby thrashing any hope of uniformity in drawing floats from $[a, b)$.

3.2. C++

The C++ standard library offers the `uniform_real_distribution()` function to draw floats from an interval $[a, b)$. The ISO/IEC 14882:2020 standard for C++ specifies only that the bounds a and b should be such that $a \leqslant b$ and $b - a \leqslant \mathsf{floatmax}(T)$, with T the type of a and b.

The implementation in GNU C++ v. 11.1.0 uses the formula:

$$y = x(b - a) + a \tag{3}$$

where x is a random float drawn from $[0, 1)$. As the example below shows, it is possible to draw the right bound for some values of a and b, in violation of the definition in the C++ standard:

```
int main(void)
{
  std::mt19937_64 rd(43);
  rd.discard(47964057);
  double a = 50000000.5;
  double b = 50000001.0;
  std::uniform_real_distribution<double> dist(a,b);
  double y = dist(rd);
  std::cout << "Right bound drawn: " << std::boolalpha
            << (y == b) << std::endl;
}
```

The output of this program is:

```
Right bound drawn: true
```

This is quite ironic since the implementation of the function `generate_canonical()`, which draws floats from $[0, 1)$, is carefully designed, in this version of the GNU C++ library, to never return its right bound (see [Goualard 2020] for an history of past failures to do so).

The web page for `uniform_real_distribution()` on cppreference.com —a respected and authoritative website about C++— warns the user that some implementations of the function may sometimes return the right bound when instantiated for single precision (the C++ `float` type). As our example above shows, this is incorrect and conflates two unrelated issues: by interpreting literally the standard, old implementations of `generate_canonical()` used to draw single precision floats in $[0, 1]$ instead of $[0, 1)$. This problem is completely unrelated to the one induced by rounding errors when computing Equation (3) with either single or double precision floats.

The same web page also suggests to call the constructor `uniform_real_distribution()` with the parameters a and $\mathsf{next}(b)$ to get floats in the interval $[a, b]$. As we will see in Section 4, rounding errors make such a workaround a misguided approach that will not work in general.

3.3. C#

The C# `RandomNextDouble()` function returns a random float in $[0, 1)$. There is no dedicated function to draw a float in an interval $[a, b)$ but the Microsoft official documentation for the `Random` class helpfully suggests four ways to do so, depending on the values of a and b:

(1) If $b = a + 1$, compute $y = a +$ `Random.NextDouble()`
(2) If $a = 0$, compute $y =$ `Random.NextDouble()` $\times\, b$
(3) If $b = 0$, compute $y =$ `Random.NextDouble()` $\times\, a$
(4) For all other cases, compute $y =$ `Random.NextDouble()` $\times\, (b - a) + a$

We will see in Section 4.1 that none of these formulas is without its flaws.

3.4. Fortran

In Fortran 90, the `RANDOM_NUMBER` function returns a float in $[0, 1)$ but there is no standard way to draw a float in an interval $[a, b)$. The RANLIB library, available on NetLib, has the `genunf()` function, which simply implements the usual affine transformation:

$$y = a + (b - a)x$$

with $x \in [0, 1)$, without checking for possible overflows.

3.5. Go

Go offers only the functions `Float32()` and `Float64()` in the `rand` package to draw single precision and double precision floats from $[0, 1)$. It is up to the user to write the code to draw from $[a, b)$.

3.6. Java

The `ThreadLocalRandom` package offers the method `nextDouble()` to draw a random float from $[a, b)$. In OpenJDK, it is implemented by first drawing a value x in $[0, 1)$ and applying the affine transformation $y = a + (b - a)x$. If y is equal to b, the method returns $\mathsf{prev}(b)$, which breaks uniformity: if $(b - a)/2^p = \underline{\gamma}(b)$, then $\mathsf{prev}(b)$ may be returned more often than it should, and if $(b - a)/2^p \neq \underline{\gamma}(b)$, the floats returned by `nextDouble()` are no longer spatially equidistributed.

3.7. JavaScript

JavaScript offers a `Math.random()` function to draw floats from $[0, 1)$ but no function to draw from $[a, b)$.

The Mozilla *SpiderMonkey* implementation of `Math.random()` divides a random integer drawn from $[0, 2^{53} - 1]$ by 2^{53} to obtain a double precision float in $[0, 1)$. To obtain a float in $[a, b)$ from a float x in $[0, 1)$, the official Mozilla documentation proposes the code for a function `getRandomArbitrary()` that implements the affine transformation $y = x(b-a)+a$. A note warns the user that, due to rounding, the function may return the right bound *"[i]f extremely large bounds are chosen (2^{53} or higher) [...]"*. This is a wrong statement, since it is easy to check that the situation may happen with small bounds too. For example, with $x = 1 - 2^{-53}$, $a = 3.5$ and $b = 4.5$, we have: $\mathsf{fl}\langle x(b-a) + a\rangle = b$.

3.8. Julia

The Random standard package draws floats uniformly at random from $[0, 1)$ by first generating a float in $[1, 2)$ and then subtracting 1.0. As a consequence, $\mathbb{S}_0^1 = \{k2^{1-p} \mid 0 \leqslant k < 2^{p-1} - 1\}$.

The `Distributions` package defines the `Uniform` distribution to draw floats from an interval $[a, b]$. However, its implementation simply applies the affine transformation $y = a + (b - a)x$ with $x \in [0, 1)$. We would therefore expect the definition to consider $[a, b)$ only. It looks like the developers of the package wanted to take into account the possibility to obtain the right bound through rounding errors, even though it introduces an unwarranted expectation from the user since some values for a and b will never allow to reach the bound b.

3.9. MATLAB

Matlab allows to draw a float x uniformly in $(0, 1)$ with the `rand()` function. The official documentation suggests to use the affine transform $y = (b - a)x + a$ to obtain a float in (a, b). A note warns the user that both bounds may occasionally be returned even though *"this is extremely unlikely to happen"*. No suggestion of being wary of overflows when a and b are large floats of opposite signs is made, though.

3.10. GNU Octave

Octave is a free open-source *clone* of Matlab. Its `rand()` function returns a single or double precision float in $(0, 1)$. The unifrnd() function from the `statistics` package returns floats drawn from $[a, b]$. Since its implementation computes a value x with `rand()` and performs the affine transformation $y = a+(b-a)x$, it is quite surprising to read in the documentation that y will belong to $[a, b]$, as being able to draw the bounds a and b themselves is entirely dependent on their actual values.

3.11. Python

The `random` standard library for Python 3 computes a double precision float x in $[0, 1)$ by using a *Mersenne Twister* [Matsumoto and Nishimura 1998] to obtain a 53 bits random integer that is multiplied by 2^{-53}. The `uniform()` function applies the affine transformation $y = a + (b - a)x$ to obtain a number in $[a, b)$. However, the documentation explicitly states that b may or may not be returned depending on the rounding.

The Numpy numerical library offers the `Random.uniform()` function to generate a float in $[a, b)$. Its implementation uses the usual affine transformation: $y = a + (b - a)x$. A note in the documentation states that *"[T]he high limit may be included in the returned array of floats due to floating-point rounding in the equation $low + (high - low) * random\_sample()$"*. The documentation is inconsistent in that it first declares that the intended domain is $[a, b)$, then proceeds to state that the value returned is lower or equal to b.

New code in Numpy uses the concept of generators. The `uniform` generator uses the same function as `Random.uniform()`, which relies on the affine transformation seen previously. However, it first checks that $b-a$ does not overflow and returns an error if it does. Contrary to the documentation for `Random.uniform()`, the documentation for the generator erroneously

claims throughout that b will never be returned, and does not warn the user about rounding errors, even though the code used is the same.

3.12. R

R provides the function `runif()` to compute values in a domain (a, b). It first checks that neither a nor b are infinite; it then draws a float x in $(0, 1)$ using rejection to ensure that none of the bound is ever drawn, and computes $y = a + (b - a)x$. The formula does not preclude y to be b and allows $b - a$ to be infinite without check.

The documentation explicitly states that neither a nor b are returned except in some conditions:

> *"runif will not generate either of the extreme values unless max = min or max−min is small compared to min, and in particular not for the default arguments."*

In all generality, the remark would be incorrect. Take for example $a = 2.0$ and $b = 4.0$. We have $b - a = a$, which means that $b - a$ is not small compared to a and still, $\mathsf{fl}\langle a + (b - a)(1 - 2^{-53})\rangle = b$, which means that the bound b can be attained. However, since the RNGs provided with R all generate double precision floats by dividing a 31 bits or 32 bits integer by 2^{31} or 2^{32}, the objection may not hold *for the standard generators provided by R* but it will hold for packages providing faster/better RNGs, such as dqRNG.

3.13. Rust

Rust offers the function `gen_range()` to draw floats from a closed interval $[a, b]$ or a half-closed interval $[a, b)$. It first generates a float in $[0, 1)$ with the function `sample_single()` using the same algorithm as Julia (Section 3.8). It checks that $b-a$ does not overflow. When generating a float in $[a, b)$, it tests that the number obtained by the affine transformation $(b - a)x + a$ is strictly smaller than b. If the value generated is greater or equal to b, the procedure is iterated to get another value.

If $s = \mathsf{fl}(b - a)$ is not finite, $\mathsf{prev}\,(s)$ is used instead (with $\mathsf{prev}\,(s) = \mathsf{floatmax}(T)$, where T is the floating-point type used), a new random float x in $[0, 1)$ is drawn and the affine transformation $\mathsf{floatmax}(T)x + a$ is performed. As a consequence, requesting values in a domain $[a, b)$ when $b - a$ overflows will always return negative values, as the following example shows:

```
use rand::Rng;
fn main() {
  let mut rng = rand::thread_rng();
  let mut i = 0;
  for _n in 1..2000000 {
      let y : f64 = rng.gen_range(-f64::MAX..=f64::MAX);
      if y >= 0.0 {
          i = i + 1;
          println!("Float: {: >+9e}", x);
      }
  }
  println!("Positive values: {}",i)
}
```

The output of the program is:

```
Positive values: 0
```

3.14. Scilab

Scilab provides the function `grand()` to draw double precision floats from $[a, b)$. The implementation first computes a float x in $[0, 1)$ and applies the affine transformation $y = a + (b - a)x$. The documentation for the uniform distribution explicitly states that the right bound b is never returned. This is incorrect, as the following Scilab commands demonstrate:

```
// We initialize the RNG to use the Mersenne Twister with a unique seed
// of 43. We then draw 3000000 random values from [a,b)
// and request the indices of the values equal to b. There are two.
--> grand('setgen','mt'); grand('setsd',43); a=3.5; b=3.5000000004656613;

--> Y=grand(3000000,1,"unf",a,b); found=find(Y==b)
found  =

  2751203.   2811196.
```

3.15. Swift

Swift offers the function `random()` to generate a float in $[a, b)$ or $[a, b]$. The implementation computes first a float x in $[0, 1)$ and then applies the affine transformation $y = (b-a)x+a$. A comment in the code acknowledges the possibility of overflow for $b - a$ and imposes the precondition that $b - a$ be finite. When drawing values in the half-closed interval $[a, b)$, the function `random()` is called recursively if y equals b. For the closed interval $[a, b]$, a bit drawn at random is used to decide whether to return b or $a + (b - a)(1 - 2^{-p})$ when the significand of x is $2^{p-1} - 1$. The documentation for `random()` recognizes the failure of the affine transformation to ensure uniformity by stating that *"[d]epending on the size and span of range, some concrete values may be represented more frequently than others."*.

Table II summarizes our findings concerning the languages and libraries surveyed. Column "**F./A.**" is "F" when a function is provided to draw a float in $[a, b)$ (or, sometimes, (a, b)) and "A" if only an algorithm is suggested; Column "**Transf.**" presents the transformation used; Column "**Ov.**" is equal to "UA" if the method used never overflows, to "UH" if no provision is made to handle overflows, to "PH" if a test is made for an overflow but no steps are taken to return a value, to "H" if an overflow is detected and correctly taken care of, and to "P" if only a precondition on $b - a$ is stated to avoid overflows; Column "**Bds.**" is equal to "Yes" if the bounds are respected —e.g., not returning b for the interval $[a, b)$—, and to "No" otherwise; Column "**Un.**" is equal to "Yes" if both spatial equidistributivity and uniformity are respected in drawing floats, and to "No" otherwise.

As can be seen at a glance from the table, no language or library ensures all the desirable properties, and many fail to ensure at least one of them.

4. ANALYZING THE IMPLEMENTATIONS

Let us now analyze theoretically the formulas we have encountered previously. In Section 3, we have seen three different algorithms to draw a float y uniformly at random in $[a, b)$. They all start with a float x drawn uniformly from $[0, 1)$ (or $(0, 1)$, for some languages). For the sake of simplicity, we will consider that $\mathbb{S}_0^1$ is the set $\{k2^{-p} \mid 0 \leqslant k < 2^p\}$, which it is for most languages and libraries. The only exceptions are those languages and libraries that consider an open interval $(0, 1)$, or that compute a random value by constructing a float in $[1, 2)$ and subtracting 1 (in that case, $\mathbb{S}_0^1$ would be $\{k2^{1-p} \mid 0 \leqslant k < 2^{p-1}\}$).

We have the formulas:

$$y = a + (b - a)x \tag{4}$$

Table II: Summarizing the characteristics of the various implementations.

Lang./Lib.	F./A.	Transf.	Ov.	Bds.	Un.
GNU C / GSL 2.7	F	$y = a(1-x) + bx$	**UA**	No	No
GNU C++ 11.1.0	F	$y = x(b-) + a$	UH	No	No
C#	A	$y = \begin{cases} a + x & \text{if } b = a+1 \\ ax & \text{if } b = 0 \\ bx & \text{if } a = 0 \\ x(b-a) + a & \text{otherwise} \end{cases}$	UH	No	No
F90/Ranlib	F	$y = a + (b-a)x$	UH	No	No
Go	—	—	—	—	—
Java OpenJDK	F	$y = a + (b-a)x$	UH	**Yes**	No
JavaScript *SpiderMonkey*	A	$y = x(b-a) + a$	UH	No	No
Julia 1.6.1	F	$y = a + (b-a)x$	UH	No	No
Matlab	A	$y = (b-a)x + a$	UH	No	No
Octave/ statistics	F	$y = a + (b-a)x$	UH	No	No
Python 3	F	$y = a + (b-a)x$	UH	No	No
Python 3/ Numpy 1.21	F	$y = a + (b-a)x$	**PH**	No	No
R 4.1.0	F	$y = a + (b-a)x$	UH	No	No
Rust 1.52.1	F	$(b-a)x + a$	**H**	**Yes**	No
Scilab 6.1.0	F	$y = a + (b-a)x$	UH	No	No
Swift 5.4	F	$y = (b-a)x + a$	**P**	**Yes**	No

Legend: "F" Function, "A" Algorithm, "UA" Unaffected, "UH" Unhandled, "PH" Partially Handled, "H" Handled, "P" Precondition.

$$y = a(1-x) + bx \tag{5}$$

$$y = a + x \qquad \text{if } b = a+1 \tag{6a}$$
$$y = bx \qquad \text{if } a = 0 \tag{6b}$$
$$y = ax \qquad \text{if } b = 0 \tag{6c}$$
$$y = x(b-a) + a \qquad \text{otherwise.} \tag{6d}$$

Floating-point arithmetic is commutative. We therefore conflate the expressions $x(b-a)+a$, $(b-a)x+a$, $a+(b-a)x$, and $a+x(b-a)$ into the one expression $a+(b-a)x$.

4.1. Analyzing Formulas (6a), (6b), and (6c)

At first glance, the expression $y = a+x$ for $x \in [0,1)$ when $b = a+1$ seems a reasonable way to compute a float y in $[a,b)$. However, we will really compute $y = \mathsf{fl}(a+x)$ with $x \in \mathbb{S}_0^1$. If $\underline{\gamma}(b)/2$ is strictly greater than —or greater or equal to, if $\mathsf{fraceven}(b)$— the expression $b - (a + (1-2^{-p}))$, then $a + (1-2^{-p})$ rounds to b, and the right bound of the interval can therefore be drawn (see Figure 5). When b is positive, this will happen as soon as the distance between b and $\mathsf{prev}(b)$ is greater than 2^{1-p}.

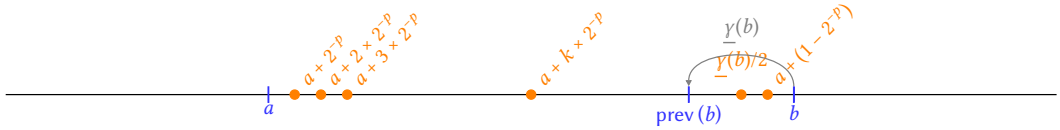


Fig. 5: When $\mathsf{fl}(a+x)=b$, for $b=a+1$.

It would seem that the expressions $y=bx$ or $y=ax$, would be subject to the same problem. As the following lemma will help to prove, the first expression respects bounds while the second does not.

LEMMA 4.1. *For any normal floating-point number c in $\mathbb{F}_p^{emax}$, we have:*

$$\begin{cases} \mathsf{fl}\langle |c|\,(1-2^{-p})\rangle = |c|\,(1-2^{-p}) & \text{if } P_2(c), \\ \mathsf{fl}\langle |c|\,(1-2^{-p})\rangle < |c|\,(1-2^{-p}) & \text{otherwise,} \end{cases} \tag{7}$$

provided no underflow occurs.

PROOF. *For clarity, we use $c'=|c|$. Let us first note that $c'(1-2^{-p})=c'-c'\times 2^{-p}$ and that $\mathsf{fl}\langle c'(1-2^{-p})\rangle=\mathsf{fl}\langle c'-c'\times 2^{-p}\rangle$ because there is only one operation that may need being (correctly) rounded on both sides (provided no underflow occurs while multiplying by 2^{-p}). We can therefore consider $\mathsf{fl}\langle c'-c'\times 2^{-p}\rangle$ instead of $\mathsf{fl}\langle c'(1-2^{-p})\rangle$.*

Assume first that c' is a power of 2. We then have: $c'=1.0\times 2^{e_c}$ and:

$$c'-c'\times 2^{-p} = \begin{array}{rl} & \overbrace{1.\ 0\ 0\ \cdots\ 0\ 0}^{p\ bits}\ 0\ 0\ 0\ \cdots\ 0\ 0 \quad \times 2^{e_c} \\ - & 0.\ 0\ 0\ \cdots\ 0\ 0\ 1\ 0\ 0\ \cdots\ 0\ 0 \quad \times 2^{e_c} \\ & 1\ 1\ 1\ \cdots\ 1\ 1 \quad \textit{(borrows)} \\ \hline & 0.\ 1\ 1\ \cdots\ 1\ 1\ 1\ 0\ 0\ \cdots\ 0\ 0 \quad \times 2^{e_c} \end{array}$$

After renormalization, we get: $\mathsf{fl}\langle c'-c'\times 2^{-p}\rangle=\overbrace{1.11\cdots 11}^{p\ bits}\times 2^{e_c-1}$, which means that $c'-c'\times 2^{-p}=\mathsf{fl}\langle c'-c'\times 2^{-p}\rangle$ The first part of the result ensues.

Considering now $c'=1.b_{p-2}b_{p-3}\cdots b_0\times 2^{e_c}$ with at least one b_i non-null, we have:

$$c'-c'\times 2^{-p} = \begin{array}{rl} & 1.\ b_{p-2}\ b_{p-3}\ \cdots\ b_1\ b_0\ 0\ 0\ 0\ \cdots\ 0\ 0 \quad \times 2^{e_c} \\ - & 0.\ 0\ 0\ \cdots\ 0\ 0\ 1\ b_{p-2}\ b_{p-3}\ \cdots\ b_1\ b_0 \quad \times 2^{e_c} \\ & 0\ \beta_{p-2}\ \beta_{p-3}\ \cdots\ \beta_1\ 1\ \mathbf{1}\ w_{p-3}\ w_{p-4}\ \cdots\ w_0 \quad \textit{(borrows)} \\ \hline & 1.\ b'_{2p-1}\ b'_{2p-2}\ \cdots\ b'_{p+1}\ b'_p\ 0\ b'_{p-2}\ b'_{p-3}\ \cdots\ b'_1\ b'_0 \quad \times 2^{e_c} \end{array}$$

Since one of the b_is is non-null, the borrow in bold red italic must be equal to 1. Consequently, the first non-storable bit of the result is a zero, which means that $c'-c'\times 2^{-p}$ must be rounded by default and not by excess. The result ensues. □

If $a=0$, we can now assert that the left bound of $[a,b)$ can be attained —when $x=0$— and that the right bound cannot, by monotonicity of rounding and thanks to Lemma 4.1. By symmetry, if $b=0$, we cannot obtain a while we can obtain b, which betrays the user expectation.

Besides the problem of reaching the bounds, there is also the problem of uniformity, or lack thereof. For the formulas (6a), (6b), and (6c), if the distance between adjacent floats

varies in $[a, b)$, the number of floats from $\mathbb{S}_0^1$ being projected after rounding to each float of $[a, b)$ will also vary.

4.2. Analyzing Formula (4)

We have already pointed out in Section 2 that Formula (4), the most often implemented formula, overflows when a and b are large floats with opposite signs.

Before investigating further, we will state two lemmas that will prove useful later on.

LEMMA 4.2. *Given $(x_1, x_2) \in \mathbb{S}_0^1 \times \mathbb{S}_0^1$ and a pair of normal floats (a, b), with $a < b$, we have:*

$$x_1 < x_2 \implies \mathsf{fl}\langle a + (b-a)x_1\rangle \leqslant \mathsf{fl}\langle a + (b-a)x_2\rangle \tag{8}$$

So, the order of the values in $\mathbb{S}_0^1$ is not changed by the affine transformation, despite rounding errors.

PROOF. *We have:*

$$\mathsf{fl}(b-a)\, x_1 < \mathsf{fl}(b-a)\, x_2$$

since $\mathsf{fl}\langle b-a\rangle$ is equal to 0 only if $a = b$, which violates the hypothesis.

By monotonicity of rounding, we get:

$$\mathsf{fl}(\mathsf{fl}(b-a)\, x_1) \leqslant \mathsf{fl}(\mathsf{fl}(b-a)\, x_2)$$

Then:

$$a + \mathsf{fl}(\mathsf{fl}(b-a)\, x_1) \leqslant a + \mathsf{fl}(\mathsf{fl}(b-a)\, x_2)$$

And finally, by monotonicity of rounding again:

$$\mathsf{fl}(a + \mathsf{fl}(\mathsf{fl}(b-a)\, x_1)) \leqslant \mathsf{fl}(a + \mathsf{fl}(\mathsf{fl}(b-a)\, x_2))$$

□

LEMMA 4.3. *Given a pair of normal floats (a, b), with $a < b$, we have:*

$$\mathsf{fl}\langle a + (b-a)x\rangle \leqslant b, \quad \forall x \in \mathbb{S}_0^1.$$

PROOF. *Thanks to Lemma 4.2, we just have to prove $\mathsf{fl}\langle a + (b-a)(1-2^{-p})\rangle \leqslant b$.*

We have:

$$\mathsf{fl}(b-a)\,(1-2^{-p}) = (b-a)(1+\delta)(1-2^{-p}), \quad \delta \in [-2^{-p}, 2^{-p}].$$

Since $(1+\delta)(1-2^{-p}) < 1$, we get:

$$\mathsf{fl}(b-a)\,(1-2^{-p}) < (b-a)$$

If $b-a$ is not a power of two, we have $\mathsf{fl}\langle (b-a)(1-2^{-p})\rangle < \mathsf{fl}(b-a)\,(1-2^{-p})$ by Lemma 4.1. Then:

$$a + \mathsf{fl}\big\langle (b-a)(1-2^{-p})\big\rangle < a + \mathsf{fl}(b-a)\,(1-2^{-p}) < a + (b-a)$$

Therefore:

$$\mathsf{fl}\big\langle a + \mathsf{fl}\big\langle (b-a)(1-2^{-p})\big\rangle\big\rangle \leqslant b$$

by monotonicity of rounding.

The proof when $b-a$ is a power of two can be done analogously and will be omitted. □

We can prove a sufficient condition on a and b for $a + (b-a)x$ to be different from b for all values x in $\mathbb{S}_0^1$:

PROPOSITION 4.4. *Given a and b two normal floats, with $a < 0$ and $a < b$, we have:*

$$a \leqslant b \times \frac{(1-2^{-p})^3}{(1-2^{-p})^3 - 1/2} \wedge a \leqslant b \times \frac{2^{-2p}-1}{1+2^{-2p}} \quad \Longrightarrow \quad a + (b-a)x \neq b, \quad \forall x \in \mathbb{S}_0^1, \tag{9}$$

provided $b \neq 0$ and no underflow occurs.

If $b = 0$, we have:

$$\forall x \in \mathbb{S}_0^1 \colon a + (b-a)x \neq b$$

PROOF. *As said previously, thanks to Lemma 4.2 we just have to give a proof for $x = (1-2^{-p})$. For $b = 0$, we can use Lemma 4.1 to prove that $\mathsf{fl}\langle a - a(1-2^{-p})\rangle$ can never be equal to b.*

The gist of the proof for $b \neq 0$ is to show that, under the conditions of Equation (9), the final addition is errorless:

$$\mathsf{fl}\big\langle a + (b-a)(1-2^{-p})\big\rangle = a + \mathsf{fl}\big\langle (b-a)(1-2^{-p})\big\rangle \tag{10}$$

because, as we have:

$$\mathsf{fl}\big\langle (b-a)(1-2^{-p})\big\rangle = (b-a)(1+\delta_1)(1-2^{-p})(1+\delta_2), \quad \text{with } \delta_1 \in [-2^{-p}, 2^{-p}], \delta_2 \in [-2^{-p}, 0], \tag{11}$$

the largest possible value for $\mathsf{fl}\langle (b-a)(1-2^{-p})\rangle$ is $(b-a)(1+2^{-p})(1-2^{-p})(1+0)$, that is $(b-a)(1-2^{-2p})$, which is strictly smaller than $b-a$. As a consequence, with the errorless addition from Equation (10), $\mathsf{fl}\langle a + (b-a)(1-2^{-p})\rangle$ must be strictly smaller than b.

To prove Equation (10), we can use the inequality just given:

$$\mathsf{fl}\big\langle (b-a)(1-2^{-p})\big\rangle \leqslant (b-a)(1-2^{-2p})$$

Through simple algebraic manipulations, it comes:

$$\begin{cases} a \leqslant b \times \frac{(1-2^{-p})^3}{(1-2^{-p})^3-1/2} & \Longrightarrow \frac{\mathsf{fl}\langle (b-a)(1-2^{-p})\rangle}{-a} \geqslant \frac{1}{2} \\ a \leqslant b \times \frac{2^{-2p}-1}{1+2^{-2p}} & \Longrightarrow \frac{\mathsf{fl}\langle (b-a)(1-2^{-p})\rangle}{-a} \leqslant 2 \end{cases}$$

Using Sterbenz's lemma [Sterbenz 1973], we deduce that the subtraction $\mathsf{fl}\langle (b-a)(1-2^{-p})\rangle - (-a)$ is errorless, which proves Equation (10) and concludes our proof. □

For the IEEE 754 binary64 double precision format ("`Float64`" in Julia, "`double`" in C, C++, and Java), Proposition 4.4 translates to the following result:

$$a < 0 \wedge \frac{a}{b} < 2 \wedge a < -b \implies a + (b-a)x \neq b, \quad \forall x \in \mathbb{S}_0^1$$

Figure 6a presents an exhaustive search of all pairs (a, b) in $\mathbb{F}_5^{15}$ verifying $\mathsf{fl}\langle a + (b-a)x\rangle = b$ for x in $\mathbb{S}_0^1$. Note that, due to overflows, there is a whole set of pairs in the upper part of the leftmost topmost quadrant that cannot be tested.

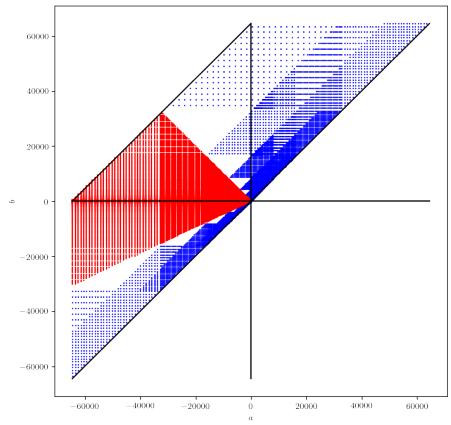
4.3. Analyzing Formula (5)

We now investigate the expression from the GSL C library:

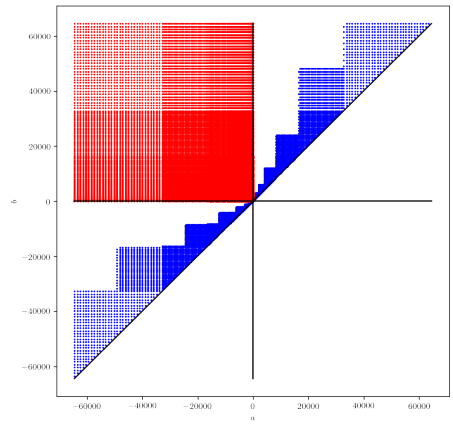
$$y = a(1-x) + bx$$

As already said, that expression has the advantage of avoiding overflows for large bounds a and b with opposite signs. It is, however, prone to *catastrophic cancellation* [Muller 2010, p. 124] when $\mathsf{fl}\langle a(1-x)\rangle \approx -\mathsf{fl}(bx)$.

Contrary to Formula (4), monotonicity is not preserved and we do not have a lemma equivalent to Lemma 4.2. We can nevertheless prove a limited version:



(a) Exhaustive search for pairs (a, b) in $\mathbb{F}_5^{15}$ such that $\mathsf{fl}\langle a + (b - a)x\rangle = b$ for x in $\mathbb{S}_0^1$. Blue dots are all pairs (a, b) such that $\mathsf{fl}\langle a + (b - a)x\rangle = b$; red dots are pairs verifying Condition (9). All eligible pairs (a, b) are between the two black slanted lines: above the higher line, the expression overflows; below the lower line, a is greater than b.



(b) Exhaustive search for pairs (a, b) in $\mathbb{F}_5^{15}$ such that $\mathsf{fl}\langle a(1 - x) + bx\rangle = b$ for x in $\mathbb{S}_0^1$. Blue dots are all pairs (a, b) such that $\mathsf{fl}\langle a(1 - x) + bx\rangle = b$; red dots are pairs (a, b) verifying Condition (13), for which we have proved that $a(1 - x) + bx < b$.

Fig. 6: Sufficient conditions for $\mathsf{fl}\langle a + (b - a)x\rangle < b$ and $\mathsf{fl}\langle a(1 - x) + bx\rangle < b$, for a and b in $\mathbb{F}_5^{15}$.

Lemma 4.5. *Given $(x_1, x_2) \in \mathbb{S}_0^1 \times \mathbb{S}_0^1$, and a and b two normal floats with $a < 0$ and $b > 0$, we have:*

$$x_1 < x_2 \implies \mathsf{fl}\langle a + (b - a)x_1\rangle \leqslant \mathsf{fl}\langle a + (b - a)x_2\rangle \tag{12}$$

So, provided $a < 0$ and $b > 0$, the order of the values in $\mathbb{S}_0^1$ is not changed by the affine transformation.

Proof. *Given $x_1 < x_2$, $a < 0$ and $b > 0$, we have:*

$$bx_1 < bx_2 \quad \text{and} \quad a(1 - x_1) < a(1 - x_2)$$

By monotonicity of rounding, we get:

$$\mathsf{fl}(bx_1) \leqslant \mathsf{fl}(bx_2) \quad \text{and} \quad \mathsf{fl}\langle a(1 - x_1)\rangle \leqslant \mathsf{fl}\langle a(1 - x_2)\rangle$$

Therefore:

$$\mathsf{fl}\langle a(1 - x_1)\rangle + \mathsf{fl}(bx_1) \leqslant \mathsf{fl}\langle a(1 - x_2)\rangle + \mathsf{fl}(bx_2)$$

And, by monotonicity of rounding again:

$$\mathsf{fl}\langle a(1 - x_1) + bx_1\rangle \leqslant \mathsf{fl}\langle a(1 - x_2) + bx_2\rangle$$

□

Under the conditions of Lemma 4.5, we may now easily analyze Formula (5) as we did for Formula (4) to find sufficient conditions for $a(1 - x) + bx$ to be strictly smaller than b for any value in $\mathbb{S}_0^1$.

Proposition 4.6. *Given two normal floats a and b, we have:*

$$a < 0 \wedge b > 0 \quad \implies \quad a(1 - x) + bx < b \quad \forall x \in \mathbb{S}_0^1 \tag{13}$$

Fig. 7: Conditions for $\mathsf{fl}\langle a(1-x)\rangle + \mathsf{fl}(bx)$ to round to b.

Proof. *Thanks to Lemma 4.5, we can consider $x = 1 - 2^{-p}$ only. Given two normal floats $a < 0$ and $b > 0$, the condition $\mathsf{fl}\langle a(1-x) + bx\rangle < b$ requires:*

$$b - \big(\mathsf{fl}\langle a(1-x)\rangle + \mathsf{fl}(bx)\big) > \underline{\gamma}(b)/2, \quad \textit{for } x = 1 - 2^{-p}$$

since it means that $\mathsf{fl}\langle a(1-x)\rangle + \mathsf{fl}(bx)$ cannot round to b (Figure 7). In truth, the strict inequality is only necessary when $\mathsf{fraceven}(b)$, but we consider this stronger constraint for the sake of simplicity.

We have:

$$b - \big(\mathsf{fl}\langle a(1-x)\rangle + \mathsf{fl}(bx)\big) = b - \big(a2^{-p} + b(1-2^{-p})(1+\delta_1)\big)$$

with $\delta_1 \in [-2^{-p}, 0]$ by Lemma 4.1. Expanding the equation, we get:

$$b - \big(\mathsf{fl}\langle a(1-x)\rangle + \mathsf{fl}(bx)\big) = (b-a)2^{-p} + b\delta_1(2^{-p} - 1)$$

From the bounds on δ_1, we deduce:

$$b - \big(\mathsf{fl}\langle a(1-x)\rangle + \mathsf{fl}(bx)\big) \geqslant (b-a)2^{-p}$$

Since we want:

$$b - \big(\mathsf{fl}\langle a(1-x)\rangle + \mathsf{fl}(bx)\big) > \underline{\gamma}(b)/2$$

we must have:

$$(b-a)2^{-p} > \underline{\gamma}(b)/2$$

Then:

$$\frac{(b-a)2^{-p}}{b} > \frac{2^{-p}2^{e_b}}{b}, \ \textit{if } \neg P_2(b)$$
$$\frac{(b-a)2^{-p}}{b} > \frac{2^{-p}2^{e_b-1}}{b}, \ \textit{if } P_2(b)$$

As b is a normal float, it is of the form $m \times 2^{e_b}$ with $1 \leqslant m \leqslant 2 - 2^{1-p}$. We can then deduce:

$$a < b \times \frac{1-2^{1-p}}{2-2^{1-p}}$$

The intersection of the set of pairs of floats (a,b) verifying both this equation and $a < 0 \wedge b > 0$ is the set of pairs (a,b) with $a < 0$ and $b > 0$. □

Figure 6b shows graphically the set of pairs (a,b) for which we can prove $\mathsf{fl}\langle a(1-x) + bx\rangle < b$ and the set of pairs (a,b) for which $\mathsf{fl}\langle a(1-x) + bx\rangle = b$. Comparing Figures 6a and 6b, Formula (5) seems to have the advantage over Formula (4), even though it is less often used[3], as it is not subject to overflows and the set of pairs (a,b) for which we can prove that the bounds are respected is larger.

[3]More accurately, it seems that only the GSL C library uses Formula (5).

5. A NEW PROCEDURE

We have seen in Section 2.2 some of the problems that plague all the formulas studied:

— The formulas generate exactly 2^p equidistributed values from $[0, 1)$ and apply a transformation to some interval that may contain more or less than 2^p equidistributed values. Factoring in the inevitable rounding errors, it is difficult to ensure an optimal generation of equidistributed values from an interval $[a, b)$ in a uniform manner;
— Rounding errors make it difficult to ensure that bounds are respected;
— Some formulas are limited by the possibility of overflows for some arguments.

Formula 4 could easily be modified to avoid overflows by writing:

$$y = 2\left(\frac{a}{2} + (\frac{b}{2} - \frac{a}{2})x\right) \tag{14}$$

instead of $y = a + (b - a)x$. Provided no underflow occurs, the two formulas are equivalent and we also have:

$$\mathsf{fl}\langle a + (b - a)x\rangle = \mathsf{fl}\left\langle 2\left(\frac{a}{2} + (\frac{b}{2} - \frac{a}{2})x\right)\right\rangle$$

Formula (14) requires a bit more computation but its performances are not much worse (See Table VI) as the division by two is very cheap. Despite its advantage, we have not observed the use of that method in the languages and libraries we surveyed.

Formula (14) only cures the problem of overflows, however. To solve all the other problems, we need a new procedure. In order to ensure spatial equidistribution, it should not use floating-point arithmetic, except where operations are guaranted to be errorless.

When a and b have the same sign and the same exponent, we can consider their fractional parts f_a and f_b as integers, draw an integer f_c in the interval $[f_a, f_b)$ and create a float with the same exponent and sign as a and b, and the fractional part f_c. Since floats with the same exponent are uniformly distributed, the procedure is sound, provided f_c is computed uniformly at random. This is a generalization of the second method described on Page 4. It is also easy to respect the bounds of the interval $[a, b)$ (and even to consider other bounds such as $(a, b), \ldots$) as no floating-point computation is involved. For example, to draw a float in (a, b), we would draw an integer f_c in $[f_a + 1, f_b - 1)$ for $b > 0$. Technically, the procedure could also be used when $[a, b] \subseteq [-2\lambda, 2\lambda]$, as all floats are spatially equidistributed in that interval, but the handling of the sign and exponent makes the method messier to implement.

If a and b do not have the same sign or the same exponent, we have to take into account the largest gap $\gamma(a, b)$ between floats in $[a, b]$ because it is impossible to get a set of spatially equidistributed floats if the smallest distance between two of them is smaller than $\gamma(a, b)$ (See Figure 8).

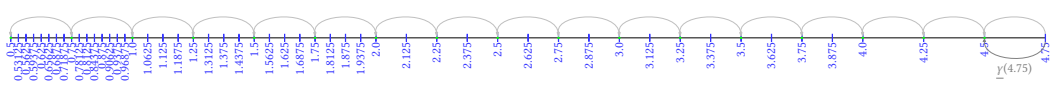


Fig. 8: Computing spatially equidistributed floats in $\mathbb{F}_4^7 \cap [0.5, 4.75]$.

A simple procedure to draw floats from an interval $[a, b)$ when $|a| \leqslant |b|$ is then to draw an integer k in $[1, \lceil (b - a)/\underline{\gamma}(b)\rceil]$ and to return $b - k\underline{\gamma}(b)$. If $b - a$ is not a multiple of $\underline{\gamma}(b)$, we have to take care of returning a whenever k is equal to $\lceil (b - a)/\underline{\gamma}(b)\rceil$.

When $|a| > |b|$, the procedure is reversed: draw an integer k in $[0, \lceil (b - a)/\overline{\gamma}(a)\rceil - 1]$ and return $a + k\overline{\gamma}(a)$.

Given g as either $\underline{\gamma}(b)$ or $\overline{\gamma}(a)$, depending on what is applicable, the division of $(b-a)$ by g should be implemented as $b/g-a/g$ to avoid an overflow. Since g is always a power of 2, the two divisions are both errorless, provided no underflow occurs. However, the final subtraction is not. In the very rare occurrences in which $\mathsf{fl}\langle b/g-a/g\rangle$ is rounded to an integer, the values $\lceil (b-a)/\underline{\gamma}(b)\rceil$ and $\lceil (b-a)/\overline{\gamma}(a)\rceil$ may be off by 1 by default. To correct that problem, we can use Dekker's exact summation algorithm [Dekker 1971] to compute $\mathsf{fl}\langle b/g-a/g\rangle$ along with an error term ε such that $\mathsf{fl}\langle b/g-a/g\rangle+\varepsilon = b/g-a/g$. If $\mathsf{fl}\langle b/g-a/g\rangle$ is not an integer, we can directly return $\lceil \mathsf{fl}\langle b/g-a/g\rangle\rceil$. Otherwise, we examine ε: if it is positive, that means that $\mathsf{fl}\langle b/g-a/g\rangle$ was rounded downward to an integer and that $\lceil b/g-a/g\rceil$ should be computed as $\lceil \mathsf{fl}\langle b/g-a/g\rangle\rceil+1$; if ε is negative or null, we can directly return $\lceil \mathsf{fl}\langle b/g-a/g\rangle\rceil$. Table III presents a Julia implementation of this algorithm.

Table III: Julia code to correctly compute $\lceil (b-a)/\underline{\gamma}(b)\rceil$ or $\lceil (b-a)/\overline{\gamma}(a)\rceil$ using Dekker's exact summation algorithm.

```
"""
    ceilint(a,b,g)

Compute ⌈b/g-a/g⌉ correctly using Dekker's algorithm for an exact summation.
"""
function ceilint(a,b,g)
   s = b/g - a/g
   if abs(a) <= abs(b)
       ε = -a/g - (s - b/g)
   else
       ε = b/g - (s + a/g)
   end
   si = ceil(Int, s)
   return (s != si) ? si : si + Int(ε > 0)
end
```

The algorithm to draw floats from an interval $[a,b)$ can easily be adapted to intervals with others bounds $((a,b),\dots)$ by changing the integer bounds of the interval we draw the value k from, in the same way as the *same exponent/same sign* algorithm above. Table IV gives naive Julia implementations for the various instances of the algorithm.

Table V compares the properties of the various algorithms presented henceforth. The equations need no explanation; *AT1PC* corresponds to the formula $a+(b-a)x$ to which we have added a test for overflow (in that case, the formula $2\big(\frac{a}{2}+(\frac{b}{2}-\frac{a}{2})x\big)$ is used instead) and a test for a violation of the open bound b (in which case we return $\mathsf{prev}\,(b)$); *AT2PC* corresponds to the formula $a(1-x)+bx$ to which we have added the same test and correction for a violation of the open bound b than AT1PC; *γ-section* is the new method we have just shown; lastly, *SESS* is the method presented at the beginning of this section when a and b have the same exponent and sign.

"*Bound respect*" is "**Yes**" when the method always respects the open right bound and never returns b; "*Spatial equidistribution*" is "**Yes**" when the set of floats $\mathbb{S}_a^b$ in $[a,b)$ that can be returned is spatially equidistributed; "*Uniformity*" is "**Yes**" when each float of $\mathbb{S}_a^b$ has the same probability to be returned; "*No overflow*" is "**Yes**" if the computation required by the method never overflows.

Note that only the new *γ-section* method is of general applicability and has a "**Yes**" in all columns. A caveat is that spatial equidistribution is guaranteed only when it is feasibly possible: if $b-a$ is not a multiple of $\gamma(a,b)$, one float of $\mathbb{S}_a^b$ will be closer to its neighbor in $\mathbb{S}_a^b$ than all other consecutive floats in $\mathbb{S}_a^b$.

Table IV: Naive Julia code to draw floats uniformly in intervals $[a, b]$, $[a, b)$, $(a, b]$, and (a, b).

```
"""
    γsectionCC(a,b)

Draw a float from an interval [a,b]
uniformly at random.
"""
function γsectionCC(a,b)
   g = γ(a,b)
   hi = ceilint(a,b,g)
   k = rand(DiscreteUniform(0,hi))
   if abs(a) <= abs(b)
       return (k == hi) ? a : b-k*g
   else
       return (k == hi) ? b : a+k*g
   end
end

"""
    γsectionCO(a,b)

Draw a float from an interval [a,b)
uniformly at random.
"""
function γsectionCO(a,b)
   g = γ(a,b)
   hi = ceilint(a,b,g)
   k = rand(DiscreteUniform(1,hi))
   if abs(a) <= abs(b)
       return (k == hi) ? a : b-k*g
   else
       return a+(k-1)*g
   end
end
```

```
"""
    γsectionOC(a,b)

Draw a float from an interval (a,b]
uniformly at random.
"""
function γsectionOC(a,b)
   g = γ(a,b)
   hi = ceilint(a,b,g)
   k = rand(DiscreteUniform(0,hi-1))
   if abs(a) <= abs(b)
       return b-k*g
   else
       return (k == hi-1) ? b : a+(k+1)*g
   end
end

"""
    γsectionOO(a,b)

Draw a float from an interval (a,b)
uniformly at random.
"""
function γsectionOO(a,b)
   g = γ(a,b)
   hi = ceilint(a,b,g)
   k = rand(DiscreteUniform(1, hi-1))
   if abs(a) <= abs(b)
       return b-k*g
   else
       return a+k*g
   end
end
```

Table V: Properties of various methods for drawing floats in $[a, b)$

Method	Bound respect	Spatial equidistribution	Uniformity	No Overflow
$a + (b - a)x$	No	No	No	No
$2\left(\frac{a}{2} + (\frac{b}{2} - \frac{a}{2})x\right)$	No	No	No	**Yes**
AT1PC	**Yes**	No	No	**Yes**
$a(1 - x) + bx$	No	No	No	**Yes**
AT2PC	**Yes**	No	No	**Yes**
γ-section	**Yes**	**Yes**†	**Yes**	**Yes**
SESS‡	**Yes**	**Yes**	**Yes**	**Yes**

† Spatial equidistribution guaranteed up to what is technically possible.
‡ *Same Exponent Same Sign*. Method limited in its applicability.

Table VI compares the performances of the methods presented, either when calling the methods to compute many random floats from the same interval $[a, b)$ ("*One call*") or when calling the methods once for many different intervals $[a, b)$ ("*Multiple calls*"). In the second case, we see that the γ-section method is more expensive because of the computation that solely depends on a and b that needs to be performed before returning a random float.

Table VI: Comparing the performances of various implementations to draw 200 000 000 floats uniformly at random from an interval $[a, b)$, either in *"One call"* for the only interval $[16, 31)$, or with *"Multiple calls"* with a and b chosen randomly in $[-10\,000, 10\,000]$ for each call.

	One call		*Multiple calls*	
Method	**binary32**	**binary64**	**binary32**	**binary64**
$a + (b - a)x$	2918	2697	9469	9150
$2\big(\frac{a}{2} + (\frac{b}{2} - \frac{a}{2})x\big)$	2748	2761	9391	9267
AT1PC†	**3045**	**2855**	**10018**	**10015**
$a(1 - x) + bx$	2797	2864	9431	9458
AT2PC†	**2792**	**2778**	**9418**	**9924**
γ-section	**2313**	**2287**	**13093**	**13237**
SESS‡	1979	2014	—	—

Times in milliseconds on an Intel i7-6700HQ @ 2.6 GHz for a C++ implementation.
†*A*ffine *T*ransformation *P*artially *C*orrected.
‡ *Same Exponent Same Sign.*

Figure 9 illustrates the benefits of using the γ-section method when drawing 1 000 000 floats from $\mathbb{F}_4^7$ in the interval $[-3.5, 1.5)$: for the first two methods, the floats are neither drawn uniformly nor spatially equidistributed, while the γ-section method offers both properties. In addition, the size of $\mathbb{S}_a^b$ for the formula $a + (b - a)x$ is only 30, down from 32 possible floats from $\mathbb{S}_0^1$ due to rounding errors and the size of $\mathbb{S}_a^b$ for $a(1 - x) + bx$ is only 31 for the same reason, while the γ-section method is capable of returning 40 different floats.

6. CONCLUSION

It seems ironic that so much care is taken by languages and their libraries to ensure that a float can be drawn uniformly at random from the interval $[0, 1)$ without ever returning 1 to fail at ensuring the same for other intervals by disregarding the properties of floating-point arithmetic. As Table II shows, none of the languages surveyed implements correctly the drawing of a float in an interval $[a, b)$. We have also found that the documentation accompanying the functions offered is often misleading in asserting the properties guaranteed.

Regarding the respect of the bounds, Propositions 4.4 and 4.6 give sufficient conditions on a and b, should a programmer elect to use the method currently implemented in ones language. All our results hold in the absence of underflow. For most applications, it is standard practice to discount the possibility of underflow in error analysis. In addition, we did not consider it worthwhile to investigate further in the event of underflows as their occurrence would require the computation of very small floats for all practical formats (values of the order of 10^{-38} for single precision and 10^{-308} for double precision).

With the double precision format, the probability of returning the value of an open bound with one of the flawed implementations we analyzed is very low (for fixed a and b, of the

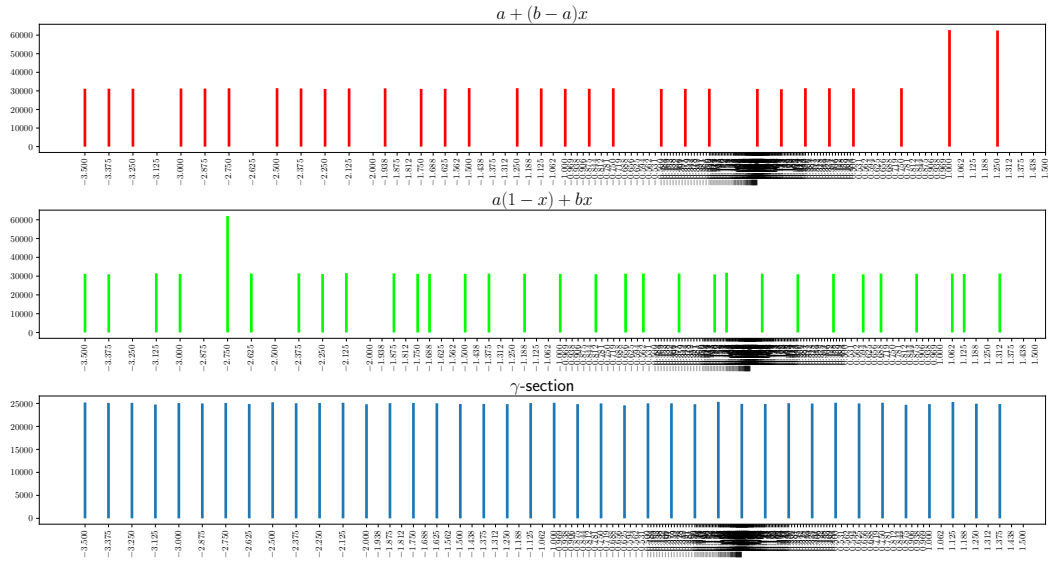


Fig. 9: Comparing the uniformity and spatial equidistributivity of three methods to draw 1 000 000 floats from $\mathbb{F}_4^7$ in $[-3.5, 1.5)$.

order of one in 10^{16} with $p = 53$). The risk is still there, and other problems plague the methods currently used, such as a lack of spatial equidistributivity and uniformity.

Table VI shows that it is possible to get spatial equidistributivity, uniformity, flexibility in the choice of the bounds, and respect of said bounds at an affordable price. For drawing multiple floats from the same interval, it is even cheaper than the flawed algorithms.

REFERENCES

T. J. Dekker. 1971. A floating-point technique for extending the available precision. *Numer. Math.* 18, 3 (June 1971), 224–242. http://link.springer.com/10.1007/BF01397083

Frédéric Goualard. 2020. Generating Random Floating-Point Numbers by Dividing Integers: a Case Study. In *Proceedings of the International Conference on Computational Science (Lecture Notes in Computer Science)*, V. Krzhizhanovskaya (Ed.), Vol. 12138. Springer, Amsterdam, The Netherlands, 15–28.

Nicholas J. Higham. 2002. *Accuracy and stability of numerical algorithms* (2nd ed ed.). Society for Industrial and Applied Mathematics, Philadelphia.

Brad Lee Holian, Ora E. Percus, Tony T. Warnock, and Paula A. Whitlock. 1994. Pseudorandom number generator for massively parallel molecular-dynamics simulations. *Phys. Rev. E* 50 (Aug 1994), 1607–1615. Issue 2.

IEEE. 2019. *IEEE Standard for Floating-Point Arithmetic*. Technical Report IEEE Std 754-2019 (Revision of IEEE 754-2008). The Institute of Electrical and Electronics Engineers, Inc.

Daniel Lemire. 2019. Fast Random Integer Generation in an Interval. *ACM Trans. Model. Comput. Simul.* 29, 1, Article 3 (Jan. 2019), 12 pages.

George Marsaglia and Wai Wan Tsang. 2004. The 64-bit universal RNG. *Statistics & Probability Letters* 66, 2 (2004), 183–187.

Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (Jan. 1998), 3–30.

Jean-Michel Muller (Ed.). 2010. *Handbook of floating-point arithmetic*. Birkhäuser, Boston.

Pat H. Sterbenz. 1973. *Floating-point computation*. Prentice-Hall, Englewood Cliffs, N.J.