Generating Random Floating-Point Numbers by Dividing Integers: a Case Study

Frédéric Goualard[0000−0002−1798−1568]

University of Nantes and LS2N UMR CNRS 6004, Nantes, France
Frederic.Goualard@univ-nantes.fr
http://frederic.goualard.net/

Abstract. A method widely used to obtain IEEE 754 binary floating-point numbers with a standard uniform distribution involves drawing an integer uniformly at random and dividing it by another larger integer. We survey the various instances of the algorithm that are used in actual software and point out their properties and drawbacks, particularly from the standpoint of numerical software testing and data anonymization.

Keywords: floating-point number · random number · error analysis.

1 Introduction

In his oft-quoted 1951 paper [18], John von Neumann asserts that "*[i]f one wants to get random real numbers on* $(0, 1)$ *satisfying a uniform distribution, it is clearly sufficient to juxtapose enough random binary digits.*" That dismissive opinion seems to have been so largely shared to this day that the implementation of methods to compute random floating-point numbers (or *floats*, for short) almost always feels like an afterthought, even in respected numerical software and programming languages. Take a look at your favorite software package documentation: chances are that it will describe the algorithm —or algorithms, since many provide more than one method— used to draw integers at random; on the other hand, it will often fail to give any precise information regarding the way it obtains random floats. Such information will have to be gathered directly from the source code when it is available. Besides, software offering the most methods to compute random integers will almost always provide only one means to obtain random floats, namely through the division of some random integer by another integer. It is the first method proposed by Knuth in *The Art of Computer Programming volume 2* [8, sec. 3.2, p. 10]; also the first method in *Numerical Recipes in C* [21, chap. 7.1, pp. 275–276] and in many other resources (e.g., [1] and [5, pp. 200–201]).

Despite its pervasiveness, that method has flaws, some of them well known, some less so or overlooked. In particular, some implementations may return values outside of the intended domain, or they may only compute values whose binary representations all share some undesirable properties. That last flaw is a baleful one for applications that rely on random number generators (RNGs) to

obtain *differential privacy* for numerical data [16], or to ensure a proper coverage in numerical software testing.

The lack of awareness to these issues may be underpinned by the fact that the libraries meant to test random number generators do not implement means to check for them properly: the procedure `ugfsr_CreateMT19937_98()` based on the Mersenne Twister [15] MT19937 to generate floats in TestU01 [10] passes the *Small Crush* battery of tests without failure, even though all double precision numbers produced have the 32^{nd} bit of their fractional part always set to "1," and their 31^{st} bit has a 75 % chance of being "0"; worse, the Kiss99 [13] generator `umarsa_CreateKISS99()` used in TestU01 passes all *Small Crush*, *Crush* and *Big Crush* batteries while the probability to be "0" of each bit of the fractional part of the numbers produced increases steadily from 0.5 for the leftmost one to the 32^{nd}, and is then equal to 1 for the remaining rightmost bits.

We briefly present in Section 2 the details of the IEEE 754 standard for binary floating-point numbers [7] that are relevant to our study. In Section 3, we will consider various implementations of the RNGs that compute random floating-point numbers through some division; that section is subdivided into two parts, depending on the kind of divisor used. The actual implementation in widely available software is considered in Section 4. Lastly, we summarize our findings in Section 5, and we assess their significance depending on the applications targeted; alternative implementations that do not use a division are also considered.

2 Floating-Point Numbers

The IEEE 754 standard [7] is the ubiquitous reference to implement binary floating-point numbers and their associated operators on processors. IEEE 754 binary floating-point numbers are of varying formats, depending on the number of bits used to represent them in memory. A format is completely defined by a pair $(p, emax)$ of two natural integers. Let $\mathbb{F}_p^{emax}$ be the set of floats with format $(p, emax)$. We will also note y_k the k^{th} bit of the binary representation of the number y (with y_0 being the rightmost least significant bit).

A floating-point number $x \in \mathbb{F}_p^{emax}$ can be viewed as a binary fractional number (the *significand*), a sign and a scale factor. There are five classes of floats: ± 0, *normal* floats with p significant bits, *subnormal* floats with less than p significant bits, *infinities* and *Not A Number*s. Only the first three classes are of any concern to us here. Floats from these classes are represented with three fields (s, E, f) with the interpretation:

$$x = \begin{cases} (-1)^s \times 1.f_{p-2}f_{p-3}\cdots f_0 \times 2^E & \text{, if } x \text{ is normal;} \\ (-1)^s \times 0.f_{p-2}f_{p-3}\cdots f_0 \times 2^{1-emax} & \text{, if } x \text{ is subnormal or zero.} \end{cases} \tag{1}$$

with s the sign bit, $E \in [1-emax, emax]$ the *exponent*, and f the *fractional part*.

The IEEE 754 standard defines three binary formats of which we will consider only the two most popular: *single precision* (aka binary32) $\mathbb{F}_{24}^{127}$ and *double precision* (aka binary64) $\mathbb{F}_{53}^{1023}$.

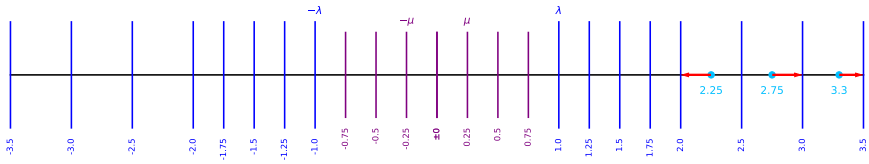

Fig. 1. The real line with $\mathbb{F}_3^1$ finite floating-point numbers. Subnormals are displayed with shorter purple segments. Reals materialized by light blue dots are rounded to nearest-even floats according to red arrows.

From Equation (1), one can anticipate the *wobbling effect* that is illustrated in Figure 1: starting with λ —the smallest positive normal float— the gap from a representable number to the next doubles every 2^{p-1} floating-point numbers, and the same goes for the negative side.

Reals in-between floating-point numbers have to be *rounded* to be represented. Given some floating-point format, let $\mathsf{fl}(x)$ be the floating-point number that is nearest to the real x. When x is at the same distance of two floats, $\mathsf{fl}(x)$ is the float whose rightmost bit b_0 of the fractional part is equal to 0. This is the *rounding to nearest-even* policy, which is usually the default and which will be assumed throughout this paper. We will also write $\mathsf{fl}\langle expr\rangle$ to denote the rounding of an expression (e.g.: $\mathsf{fl}\langle a + b \times c\rangle = \mathsf{fl}(\mathsf{fl}(a) + \mathsf{fl}(\mathsf{fl}(b) \times \mathsf{fl}(c)))$ for $(a, b, c) \in \mathbb{R}^3$).

3 Dividing Random Integers

"If you want a random float value between 0.0 and 1.0 you get it by an expression like x = rand()/(RAND_MAX+1.0)." This method, advocated here in *Numerical Recipes in C* [21, pp. 275–276], is used in many libraries for various programming languages to compute random floating-point numbers with a standard uniform distribution, the variation from one library to the next being the algorithm used to compute the random integer in the numerator and the value of the fixed integer as denominator.

For a floating-point set $\mathbb{F}_p^{emax}$, there are $emax \times 2^{p-1}$ floats in the domain $[0, 1)$, to compare with the $(2emax + 1)2^p$ finite floats overall; that is, almost one fourth of all finite floats are in $[0, 1)$. Dividing a random nonnegative integer a by an integer b strictly greater than a can only return at most b distinct floats, less than that if two fractions round to the same float. Let $\mathbb{D}_b$ be the set $\{\mathsf{fl}\langle x/b\rangle \mid x = 0, 1, \ldots, b-1\}$ with b a strictly positive integer. Two necessary and sufficient conditions for $\mathsf{fl}\langle x/b\rangle$ to be equal to x/b for any x in $\{0, 1, \ldots, b-1\}$ are that b be:

1. a power of 2, otherwise some x/b are bound to not be dyadic rationals;
2. smaller or equal to 2^p, since not all integers greater than 2^p are representable in $\mathbb{F}_p^{emax}$. Besides, the largest gap between two consecutive floats from $[0, 1)$ being 2^{-p}, x/b might not be representable for the large values of x otherwise.

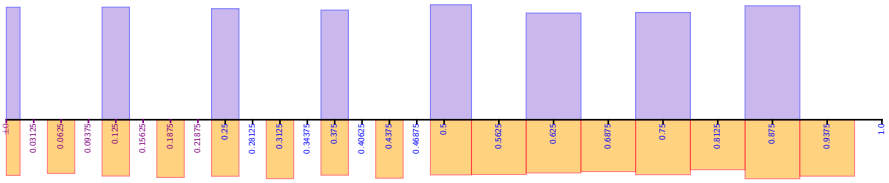

Fig. 2. Drawing 10 000 floats at random from $\mathbb{F}_4^3$ by drawing random integers from $[0, 7]$ and dividing by 8 (top purple bars), or by drawing random integers from $[0, 15]$ and dividing by 16 (bottom orange bars).

All members of $\mathbb{D}_b = \{k/b \mid k = 0, \ldots b-1\}$, for any b in $\{2^i \mid i = 0, \ldots, p\}$, are then representable in $\mathbb{F}_p^{emax}$ and uniformly distributed on $[0, 1)$. Figure 2 shows such situations on $\mathbb{F}_4^3$: the top purple bars correspond to drawing $10,000$ random integers from $[0, 7]$ and dividing them by 2^3, while the bottom orange bars correspond to drawing the same number of integers from $[0, 15]$ and dividing them by $2^p = 16$. Rectangle heights indicate the number of times the corresponding float was obtained; the width of each rectangle indicates the set of reals that would all round to that same floating-point value. We see in Figure 2 that the floats obtainable are computed with the same frequency, and that they split neatly the real line on $[0, 1)$ into eight (resp. sixteen) same-sized segments.

To ensure a uniform spreading on $[0, 1)$, we have to meet the two abovementioned conditions, the second one of which means that we cannot expect to be able to draw uniformly more than a fraction $2^p/(emax \times 2^{p-1}) = 2/emax$ of all the floats in $[0, 1)$. To put that in perspective, it means that, for the double precision format $\mathbb{F}_{53}^{1023}$, we can only draw less than $0.2\,\%$ $(2/1023)$ of all floating-point numbers in $[0, 1)$.

If we violate one of the two conditions, the set $\mathbb{D}_b$ contains floats that are no longer spread evenly on $[0, 1)$ (see the examples in Figure 3). When $b \leqslant 2^p$, each float in $\mathbb{D}_b$ has the same probability of being chosen (since then, no two values a_1/b and a_2/b can round to the same float when $a_1 \neq a_2$). On the other hand, when $b > 2^p$, several fractions a/b may round to the same float, leading to it being over-represented. This is the case as soon as the distance $2^E 2^{1-p}$ between two consecutive floats is greater than b^{-1} (See Figure 3 for $b = 24$).

As previously pointed out, the gap between adjacent floats increases from 0 to 1, with the largest gap being 2^{-p} between 1 and $\mathsf{prev}\,(1)$, its predecessor. We have seen that a division-based method to compute random floats that are evenly spread on $[0, 1)$ cannot offer more than a very small proportion of all the floats in that domain. In particular, if $emax$ is not smaller than p, it is not possible to draw subnormal floats by division while preserving uniform spreading of the floats we can draw. Provided we use a procedure that draws integers in $[0, b-1]$ uniformly at random —an assumption we will always make in this paper—, all floats from $\mathbb{D}_b$ are also drawn uniformly at random for $b \leqslant 2^p$. This

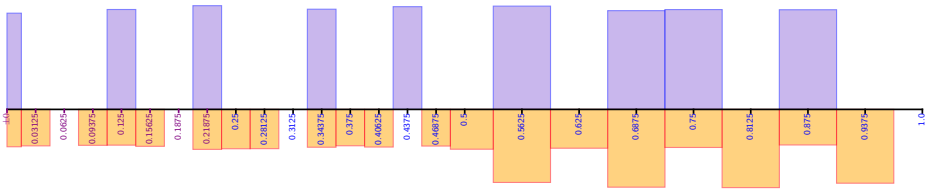

Fig. 3. Drawing 10 000 floats at random from $\mathbb{F}_4^3$ by drawing random integers in $[0, 8]$ and dividing by 9 (top purple), and by drawing from $[0, 23]$ and dividing by 24 (bottom orange).

is different from claiming that the division-based procedure draws the floats from $[0, 1)$ uniformly at random since only one in $2/emax$ can ever be drawn. We must also keep in mind that it is also markedly different from claiming that the floats computed are the rounded results of drawing real values in $[0, 1)$ uniformly at random: since the space between floats doubles regularly from 0 to 1, the floats in $\mathbb{D}_b$ should be drawn following a geometric distribution for their exponent to accurately represent that. Several works [3,4,24] and [17, Chap. 9] have investigated means to do it, even though some of their authors are not yet convinced that there are real applications waiting for their method.

After analyzing a set of prominent libraries and programming languages, we have identified essentially two classes of division-based methods to compute floating-point numbers with a standard uniform distribution:

1. Division by a power of 2;
2. Division by a Mersenne number.

3.1 Dividing by a Power of 2

Since Lehmer's work on *Linear Congruential Generators* [11], many methods have been devised to compute random integers in $[0, m-1]$ using modular arithmetic with some modulus m. Initially, it was very convenient to choose m as a power of 2, preferably matching the size of the integers manipulated —since the modulo was then essentially free— even though that choice may not offer the largest period for the RNG [6]. People quickly proceeded to divide by m the integers obtained to get random numbers in $[0, 1)$ [9], which is also very convenient when m is a power of 2 as the division then amounts to a simple manipulation of the float's exponent.

To this day, many libraries still draw floats by dividing random integers by a power of 2 to obtain random floats in $[0, 1)$ or $(0, 1)$ in order to get the benefit of avoiding a costly "real" division.

For example, the canonical method to get random floats in $[0, 1)$ for the ISO C language is to call the `rand()` function, which returns a random integer in

$[0, \texttt{RAND\_MAX}]$, and to divide it by `RAND_MAX+1` (see, e.g., [21] and [23, Q. 13.21]), even though one of its flaws is well known: according to Section 7.22.2.1 of the *ISO/IEC 9899:2011 standard for C*, `RAND_MAX` is only constrained to be at least $2^{15} - 1$, and the GNU C library sets it to $2^{31} - 1$ only [12]. As shown above, the theoretical maximum value of the denominator to get uniformly spaced double precision floats in $[0, 1)$ is 2^{53}. It is then wasteful to only divide by 2^{15} or even 2^{31}, leading to very few possible floating-point numbers.

A property expected from RNGs that compute integers is that each bit of their binary representation be equally likely a "0" or a "1." The left graph in Figure 4 shows the probability computed by drawing 1 000 000 integers in $[0, 2^{31} - 1]$ using a Mersenne Twister-based RNG. We see that all bits from the zeroth in the rightmost position to the thirtieth in the leftmost position have a probability approximately equal to 0.5 to be a "1," which is expected.

However, when we divide these integers from $[0, 2^{31} - 1]$ by 2^{31} to get floats in the domain $[0, 1)$, we lose that property. The picture on the right in Figure 4 shows the probability to be "1" of each bit of the fractional part of double precision floats obtained by such a process: we see that the 22 rightmost bits of the fractional part have a probability equal to 0 to be a "1." The probability then increases steadily until it peeks at around 0.5 for the leftmost bits.

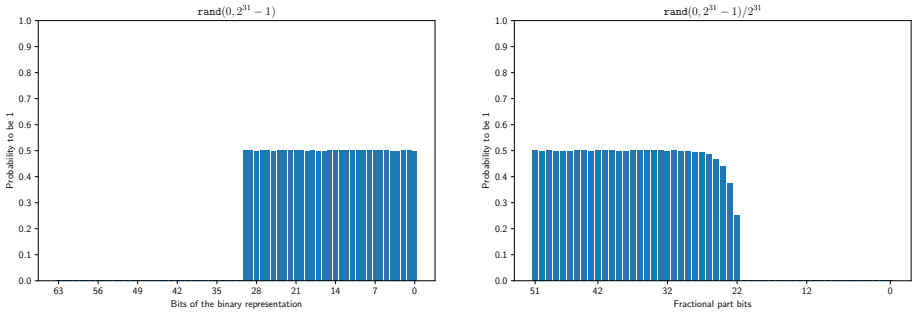

Fig. 4. Probability for each bit to be 1 (left: when drawing an integer in $[0, 2^{31} - 1]$; right: when drawing a double precision float by dividing an integer in $[0, 2^{31} - 1]$ by 2^{31}). Simulation with 1 000 000 draws.

Indeed, if we note $P[e]$ the probability of the event e, we have:

Proposition 1. *Given $\mathbb{F}_p^{emax}$ a set of floats and k an integer in $[1, p]$, let $x = x_{k-1}x_{k-2}\ldots x_1x_0$ be a strictly positive random integer in $[1, 2^k - 1]$ with $P[x_i = 1] = \frac{1}{2}$ for $i = 0, 1, \ldots, k-1$. Given $y = y_{p-1}.y_{p-2}\cdots y_0$ the significand of the*

normal float $\frac{x}{2^k}$*, with* $y \in \mathbb{F}_p^{emax}$*, we have:*

$$\begin{cases} P[y_i = 1] = 0 & \forall i \in [0, p-k-1] \\ P[y_i = 1] = \displaystyle\sum_{j=0}^{i-p+k} \frac{1}{2^{j+2}} = \frac{1}{2} - 2^{p-k-i-2} & \forall i \in [p-k, p-1] \end{cases} \tag{2}$$

Proof. The division by a power of 2 *corresponds to a simple shift of the binary point. Considering a set of floats* $\mathbb{F}_p^{emax}$*, if we draw a non-null integer* $x = x_{k-1}x_{k-2}\dots x_1x_0$ *in the domain* $[1, 2^k - 1]$ *(with* $k \leqslant p$*) and divide it by* 2^k*, we get:*

$$\frac{x}{2^k} = 0.x_{k-1}x_{k-2}\dots x_1x_0$$

That number needs to be normalized to be stored as an IEEE 754 float, and we get its significand y:

$$\begin{aligned} y &= 1.x_{k-2}\dots x_0 \underbrace{0\cdots0}_{p-k} && \text{if } x_{k-1} = 1 \\ &= 1.x_{k-3}\dots x_0 \underbrace{0\cdots0}_{p-k+1} && \text{if } x_{k-1} = 0 \wedge x_{k-2} = 1 \\ &= \quad \ddots \\ &= 1.x_{k-(j+1)}\dots x_0 \ \underbrace{0\cdots0}_{p-k+j-1} && \text{if } (\forall l \in [1, j+1] \colon x_{k-l} = 0) \wedge x_{k-j} = 1 \end{aligned}$$

Obviously, we get $P[y_i = 1] = 0$ *for all* $i \in [0, p-k-1]$*. In addition:*

$$\begin{aligned} P[y_{p-k} = 1] &= P[x_{k-1} = 1] \times P[x_0 = 1] \\ P[y_{p-k+1} = 1] &= P[x_{k-1} = 1] \times P[x_1 = 1] + \\ &\qquad P[x_{k-1} = 0] \times P[x_{k-2} = 1] \times P[x_0 = 1] \\ P[y_{p-k+2} = 1] &= P[x_{k-1} = 1] \times P[x_2 = 1] + \\ &\qquad P[x_{k-1} = 0] \times P[x_{k-2} = 1] \times P[x_1 = 1] + \\ &\qquad P[x_{k-1} = 0] \times P[x_{k-2} = 0] \times P[x_{k-3} = 1] \times P[x_0 = 1] \\ &\vdots \\ P[y_{p-k+j} = 1] &= \sum_{l=0}^{j} \prod_{i=1}^{l} \big(P[x_{k-i} = 0]\big) \times P[x_{k-(l+1)} = 1] \times P[x_{j-l} = 1] \end{aligned}$$

Considering that $P[x_i = 0] = P[x_i = 1] = \frac{1}{2}$ *for all* $x \in [1, 2^k - 1]$*, the result ensues.* □

If the value for k is greater than p, some values in $\mathbb{D}_b$ will need rounding, and the *round-to-nearest-even* rule will slightly favor results with a "0" as rightmost bit of the fractional part, as can be readily seen in Figure 5.

Another problem when k is greater than p is that $x/2^k$ may round to 1, even though we claim to compute floats in $[0, 1)$. This is what happens in software that divides a 32 bits random integer x by 2^{32} to get a single precision float (see Section 4): as soon as $x/2^{32}$ is greater or equal to $1 - 2^{-25}$, the fraction rounds to 1 when rounding to nearest-even.

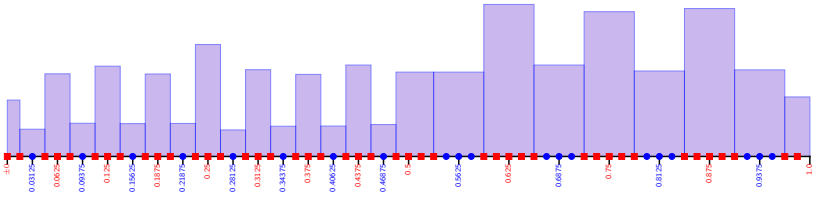

Fig. 5. Drawing 10 000 integers uniformly at random in $[0, 2^6 - 1]$ and dividing them by 2^6 in $\mathbb{F}_4^3$. Values $i/2^6$ (for $i = 0, 1, \ldots, 2^6 - 1$) are represented before rounding as dots or squares on the real line, where the color and form correspond to the float they round to.

3.2 Dividing by a Mersenne Number

The division of a random integer in $[0, m-1]$ by m gives a float in $[0, 1)$ (with 1 excluded, provided m is not too large, as seen above). To obtain values in $[0, 1]$ instead, one can divide by $m - 1$. Since m is often a power of 2 for the reasons explained in Section 3.1, the divisor is now a Mersenne number. That algorithm is proposed in many sources and in particular by Nishimura and Matsumoto —the authors of the Mersenne Twister— with a contribution by Isaku Wada [19,20]. Due to the popularity of the Mersenne Twister, their work on drawing floats has been integrated in many numerical software *as is*, along with Mersenne Twister-based RNGs for integers.

Nishimura *et al.* propose two versions of their algorithm to compute double precision floats in $[0, 1]$: one that uses the 32 bits version of MT19937 [19] to compute a 32 bits integer that is divided by $2^{32} - 1$, and the other that uses the 64 bits version of MT19937 [20] to compute a 53 bits integer that is divided by $2^{53} - 1$. In order to avoid some costly division, both versions replace the division with a multiplication by the inverse (which is precomputed at compile time).

The problem with both methods is that $\mathsf{fl}\langle 1/(2^k - 1)\rangle$ has a special structure that may induce some correlation between bits of the fractional part of the floats produced and a non-uniform probability of each bit to be "1." Indeed, we have:

Proposition 2. *Given a set of floats $\mathbb{F}_p^{emax}$ and an integer $k \in [2, p]$:*

$$\mathsf{fl}\left\langle \frac{1}{2^k - 1} \right\rangle = \begin{cases} \left(1 + 2^{1-p} + \sum_{i=1}^{\lfloor \frac{p}{k} \rfloor - 1} 2^{-ik}\right) \times 2^{-k}, & \text{if } p \equiv 0 \pmod{k}, \\ \left(1 + \sum_{i=1}^{\lfloor \frac{p}{k} \rfloor} 2^{-ik}\right) \times 2^{-k}, & \text{otherwise.} \end{cases} \tag{3}$$

Proof. We have:

$$2^k - 1 = \underbrace{11\cdots1}_{k}$$

Then:

$$\begin{aligned} \tfrac{1}{2^k-1} &= 0.\underbrace{0\cdots0}_{k-1}1\underbrace{0\cdots0}_{k-1}1\cdots \\ &= \left(1 + \textstyle\sum_{i=1}^{\infty} 2^{-ik}\right) \times 2^{-k} \end{aligned}$$

If p is a multiple of k, the first bit outside the stream of bits we can represent in the fractional part is a "1," which means we have to round upward to represent $\mathsf{fl}\langle 1/(2^k-1)\rangle$*:*

$$\frac{1}{2^k-1} = \overbrace{1.\underbrace{0\cdots0}_{k-1}1\underbrace{0\cdots0}_{k-1}}^{p}1\underbrace{0\cdots0}_{k-1}1\cdots\times 2^{-k}$$

Otherwise, we must round downward:

$$\frac{1}{2^k-1} = \overbrace{1.\underbrace{0\cdots0}_{k-1}1\underbrace{0\cdots0}_{k-l}}^{p}\underbrace{0\cdots0}_{l}1\underbrace{0\cdots0}_{k-1}1\cdots\times 2^{-k}$$

□

For $k=32$ —the first method by Nishimura *et al.* [19]—, we get:

$$\mathsf{fl}\left\langle\frac{1}{2^{32}-1}\right\rangle = 1.\underbrace{0\cdots0}_{31}1\times 2^{-32}$$

When multiplying $x = x_{31}\cdots x_0$ by $\mathsf{fl}\left\langle\frac{1}{2^{32}-1}\right\rangle$, we get the real z:

$$z = x\times\mathsf{fl}\left\langle\frac{1}{x^{32}-1}\right\rangle = 0.x_{31}\cdots x_0x_{31}\cdots x_0$$

Normalizing the result, we get:

$$\begin{aligned}
\text{if } x_{31}=1\text{:}\quad & z = 1.\overbrace{x_{30}\cdots x_0 1 x_{30}\cdots x_{11}}^{52}x_{10}\cdots x_0\times 2^{-1}\\
\text{if } x_{31}=0\wedge x_{30}=1\text{:}\quad & z = 1.\underbrace{x_{29}\cdots x_0 01 x_{29}\cdots x_{10}}_{52}x_9\cdots x_0\times 2^{-2}\\
&\cdots
\end{aligned}$$

Notice that:

- Bit 20 of the fractional part of the restriction to double precision of z is always equal to 1;
- The probability of being "1" of Bits 21 through 51 follows the same law as in Prop. 1;
- Bits of the fractional part are highly correlated since some bits of x occur twice (e.g., when $x_{31}=1$, x_{30} occur as z_{51} and z_{19}, and so on).

The first two properties can readily be seen in Figure 6.

The second method, which divides integers by $2^{53}-1$ exhibits a different behavior since then $k=p$ and, according to Proposition 2, the rounded value of $1/(2^{53}-1)$ has a different structure, viz.:

$$\mathsf{fl}\left\langle\frac{1}{2^{53}-1}\right\rangle = 1.\underbrace{0\cdots0}_{51}1\times 2^{-53}$$

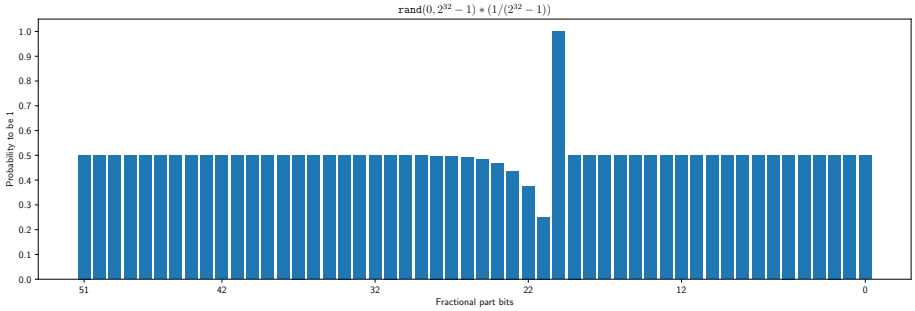

Fig. 6. Computing 1 000 000 random double precision floats in $[0, 1]$ according to the MT19937-32-based procedure by Nishimura *et al.* [19].

Note how there are now only $k-2$ "0s" in the fractional part instead of $k-1$ for $\mathsf{fl}\left\langle \frac{1}{2^{32}-1} \right\rangle$. As a consequence, there is some overlap of the bits during the multiplication by $x = x_{52} \cdots x_0$, and we get:

$$z = x \times \mathsf{fl}\left\langle \frac{1}{2^{53}-1} \right\rangle = 0.x_{52} \cdots x_1 (x_0 + x_{52}) x_{51} \cdots x_0$$

That structure of z seems sufficient to remove the most glaring flaws compared to the first method (See Figure 7 on the right). Unfortunately, we have not yet been able to explain the slight dip in the probability of bits 2 to 5 to be "1."

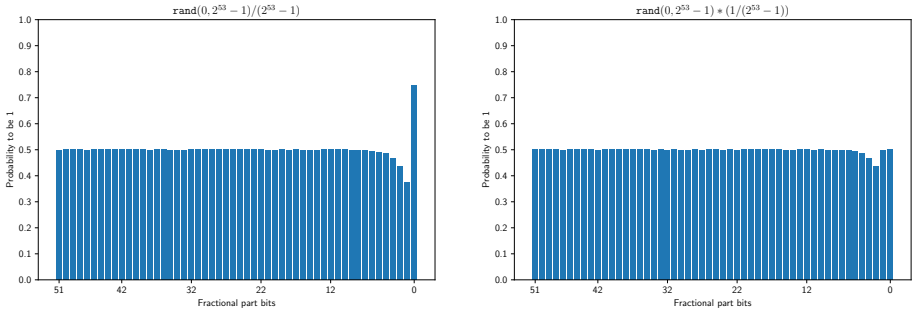

Fig. 7. Division by a Mersenne number vs. multiplication by its inverse

Since $1/(2^{53}-1)$ needs rounding to double precision, there is a difference in dividing x by $2^{53}-1$ or multiplying it by $\mathsf{fl}\langle 1/(2^{53}-1)\rangle$ as Nishimura *et al.* do. That difference also shows in the probability to be "1" of each bit of

the fractional part of the resulting floats (compare the left and right graphics in Figure 7). Since we never encountered implementations that used directly a division by a Mersenne number, we have not investigated the difference further.

Lastly, note that the division by a Mersenne number, as well as the multiplication by its inverse, involve some rounding of the result. As a consequence, both methods may introduce some non-uniformity when drawing floats, even when b is smaller or equal to p.

4 Implementations in Software

We have studied the implementation of floating-point RNGs in some major programming languages and libraries. Except when explicitly stated otherwise, we focus here on the RNGs that compute double precision floats in $[0, 1)$. Due to the popularity of the Mersenne Twister [15], many software implement one of the algorithms proposed by Nishimura, Matsumoto and Wada [19,20], of which we have shown some of the flaws in the preceding sections. Some few software do not resort to a division by a power of 2 or by a Mersenne number; they are discussed only in the last section.

We have seen in Section 3.1 that the ISO C language does not offer any standard function to compute random floats in $[0, 1)$, the accepted algorithm being to divide the result of `rand()` by `RAND_MAX+1`, where `RAND_MAX` is only $2^{31} - 1$ in the *GNU Compiler Collection* (GCC) library. This is also the approach used in the Lua language, whose library is based on C.

The GNU Scientific Library (GSL) [5] strives to be the go-to numerical library for C in the absence of a true standardized one. Unfortunately, the default implementation to compute a random float is not much better than using `rand()` as it uses a Mersenne Twister to compute a 32 bits random integer and multiply it by 2^{-32}. The same method is also used by Scilab 6.0.2.

The C++ language has been offering random number generation in its standard library since the C++11 standard. The `generate_canonical()` function is used to draw uniformly at random a floating-point number in $[0, 1)$. Unfortunately, from the C++11 standard up to the latest draft for the C++20 standard included [22, p. 1156], the algorithm mandated to compute a random float requires to divide a random integer by a value that may be much larger than 2^p, with p the size of the significand of the type of floats produced. As a consequence, the value 1.0 may be returned, in violation of the definition of the function itself. This is a known error that has been addressed formally by the *C++ Library Working Group* in 2017 only by proposing to recompute a new random float whenever the one computed is equal to 1.0. The C++ library for GCC had already implemented that workaround in its 6.1.0 release in 2015. Oddly enough, it was changed in 2017 for the 7.1.0 release in favor of returning $\mathsf{prev}(1)$ in that case, which breaks the uniformity requirement, even if only slightly.

The Go language, at least in its most current version as of 2019, generates a double precision number in $[0, 1)$ by computing a 63 bits random integer and

dividing it by 2^{63}. As a consequence, the implementation of its `Float64()` function is plagued by the same problems as C++ `generate_canonical()`, and the current method to avoid returning 1.0 is to loop generating a new number until getting a value different from 1.0. Since 63 is larger than $p = 53$, there are also uniformity problems in drawing floats, as seen in the preceding sections.

In Java, The `Random.nextDouble()` method to obtain a random double precision float in $[0, 1)$ is implemented, both in OpenJDK and in the Oracle JDK, by computing a 53 bits random integer and multiplying it by 2^{-53}. An older method used to computed a 54 bits integer and multiply it by 2^{-54}; ironically, it was discarded in favor of the new method because it was discovered that, due to rounding, it introduced a bias that skewed the probability of the rightmost bit of the significand to be "0" or "1," even though the new method presents the same bias for different reasons, as seen previously. The Rust language does it in exactly the same way. It is also the method used in both the standard library for Python, as well as in the Numpy package.

GNU Fortran 9.2.0 introduces a twist in the computation of a random double precision number in that it uses the *xoshiro256*** algorithm by Blackman and Vigna [2] to compute a 64 bits random integer in $[0, 2^{64} - 1]$, forces the 11 lowest bits to 0 to get an integer that is a multiple of 2^{11}, then divides the result by 2^{64} to get multiples of 2^{-53}. As a result, all values are representable and uniformly spread in $[0, 1)$. The same effect would be obtained by computing a 53 bits random integer and dividing it by 2^{53}. Consequently, the algorithm suffers from the flaws shown in Section 3.1.

The default algorithm in MATLAB [14] computes a double precision float in $(0, 1)$, not $[0, 1)$, by creating a 52 bits random integer, adding 0.5 to it, and dividing the sum by 2^{52} to obtain a number in $[2^{-53}, 1 - 2^{-53}]$. Evidently, that algorithm shares the same flaws as the algorithm that directly divides an integer by 2^{53}. GNU Octave, a MATLAB-like software, computes both single precision and double precision random numbers in $(0, 1)$. Single precision floats are generated by computing a 32 bits integer, adding 0.5 and dividing the sum by 2^{32}. Since the divisor is greater than 2^{23}, the distribution is no longer uniform, due to rounding. Additionally, the RNG may return exactly 1.0 in violation of its specification. The implementation of the double precision generator seems also flawed, at least in the latest version as of 2019 (5.1.0), as it consists in generating a 53 bits integer, adding 0.4 [sic] (which has to be rounded) and dividing the result by 2^{53}.

5 Conclusion

The motivation for this work stems from our attempt to study empirically the behavior of some complex arithmetic expression in the Julia language: while evaluating the expression numerous times with random floats, we discovered that the number of results needing rounding was much smaller than anticipated. Only when studying the structure of the floats produced by the `rand()` Julia function did we notice that the least significant bit of their fractional part was

consistently fixed to 0 when requiring floats in some domains but not in others. Investigating the RNGs producing floating-point numbers in other programming languages and libraries, we found they were almost all lacking in some respect when used to test numerical code.

Many studies are devoted to the analysis of RNGs producing integers; they are much fewer to consider RNGs producing floats, and we are not aware of other works considering the structure of the floats produced at the bit level. In truth, such a dearth of work on that subject might lead one to wonder whether the systemic irregularities in the structure of the fractional part produced when using a division really matter. We have identified two domains when it seems to be the case:

- When investigating numerical software empirically (see our war story directly above). In addition, the inability of the division-based methods to generate subnormals may be redhibitory for some applications;
- When anonymizing numerical data.

That last use case should take on more and more importance in the *Big Data* era, and problems with floating-point random number generators have already been reported [16], particularly when the probabilities to be "1" or "0" of the bits of the fractional part are skewed.

Some programming languages and libraries do not use a division to compute random floats. Amusingly enough, it is the case with Julia, the very language that prompted our investigation, even though its algorithm exhibits other flaws. After some experimentation, we believe it is also the case of Mathematica, at least in its version 12 (anecdotally, its default implementation exhibits the same flaws as Julia's). We have already started a larger study on the other classes of algorithms to generate random floating point numbers in the hope of isolating the better algorithms that offer both good performances, uniformity, and a perfect regularity of the structure of the fractional part of the floats produced.

Acknowledgments. We would like to thank Dr. Alexandre Goldsztejn for providing us with the means to test the random engine of Mathematica v. 12.

References

1. Beebe, N.H.F.: The Mathematical-Function Computation Handbook: Programming Using the MathCW Portable Software Library. Springer International Publishing (2017)
2. Blackman, D., Vigna, S.: Scrambled linear pseudorandom number generators (Aug 2019), https://arxiv.org/abs/1805.01407, revision 2
3. Campbell, T.R.: Uniform random floats (Apr 2014), https://mumble.net/~campbell/2014/04/28/uniform-random-float, unpublished note
4. Downey, A.B.: Generating pseudo-random floating-point values (Jul 2007), http://allendowney.com/research/rand, unpublished note
5. Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Alken, P., Booth, M., Rossi, F., Ulerich, R.: The Gnu Scientific Library documentation. https://www.gnu.org/software/gsl/doc/html/ (Aug 2019)

6. Hull, T., Dobell, A.: Random number generators. SIAM Review **4**(3), 230–254 (1962)
7. IEEE Standards Association: IEEE standard for floating-point arithmetic. IEEE Standard IEEE Std 754-2008, IEEE Computer Society (2008)
8. Knuth, D.E.: The art of computer programming: seminumerical algorithms, vol. 2. Addison-Wesley Professional, third edn. (1997)
9. L'Ecuyer, P.: History of uniform random number generation. In: 2017 Winter Simulation Conference (WSC). pp. 202–230 (Dec 2017)
10. L'Ecuyer, P., Simard, R.: TestU01: A C library for empirical testing of random number generators. ACM Trans. Math. Softw. **33**(4), 22:1–22:40 (Aug 2007)
11. Lehmer, D.H.: Mathematical methods in large-scale computing units. In: Proceedings of the Second Symposium on Large Scale Digital Computing Machinery. pp. 141–146. Harvard University Press, Cambridge, United Kingdom (1951)
12. Loosemore, S., Stallman, R., McGrath, R., Oram, A., Drepper, U.: The GNU C library reference manual for version 2.30 (2019)
13. Marsaglia, G.: Random numbers for C: The END? Post at sci.stat.math and sci.math (Jan 1999), available at http://www.ciphersbyritter.com/NEWS4/RANDC.HTM#36A5FC62.17C9CC33@stat.fsu.edu
14. MathWorks: MATLAB documentation: Creating and controlling a random number stream. Web page at https://www.mathworks.com/help/matlab/math/creating-and-controlling-a-random-number-stream.html (2019), retrieved 2019-12-28
15. Matsumoto, M., Nishimura, T.: Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. **8**(1), 3–30 (Jan 1998)
16. Mironov, I.: On significance of the least significant bits for differential privacy. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. pp. 650–661. CCS '12, ACM, New York, NY, USA (2012)
17. Moler, C.: Numerical Computing with MATLAB. SIAM (2004)
18. von Neumann, J.: Various techniques used in connection with random digits. National Bureau of Standards Applied Mathematics Series **12**, 36–38 (1951)
19. Nishimura, T., Matsumoto, M.: A C-program for MT19937, with initialization improved 2002/1/26. Available at http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/CODES/mt19937ar.c (Jan 2002), (with a contribution by Isaku Wada). Retrieved 2019-12-30
20. Nishimura, T., Matsumoto, M., Wada, I.: A C-program for MT19937-64 (2004/9/29 version). Available at http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/VERSIONS/C-LANG/mt19937-64.c (Sep 2004), retrieved 2019-12-30
21. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C. Cambridge University Press, second edn. (1992)
22. Smith, R., Köppe, T., Maurer, J., Perchik, D.: Working draft, standard for programming language c++. Tech. Rep. N4842, ISO/IEC (Nov 2019), committee draft for C++20
23. Summit, S.: C programming FAQs: Frequently Asked Questions. http://c-faq.com/, retrieved 2019-12-21
24. Walker, A.J.: Fast generation of uniformly distributed pseudorandom numbers with floating-point representation. Electronics Letters **10**(25), 533–534 (December 1974)