The Schubfach way to render doubles

Raffaello Giulietti

2021-11-07

1 Introduction

This writing is mainly about rendering double values as decimal strings. What is the problem addressed here? While each finite double is also a decimal (§3.2), it turns out that its expansion is almost always very long. For example, the full decimal expansion of the humble looking Java double 1.2 is

$$1.1999999999999999555910790149937383830547332763671875 \tag{1}$$

a monster with 53 digits!

How come? No double is exactly 1.2 (§3.1). In Java, the literal 1.2 really stands for the double *closest* to the decimal, namely $\mathsf{0x13\,3333\,3333\,3333} \cdot 2^{-52} = 5\,404\,319\,552\,844\,595 \cdot 2^{-52}$, which indeed happens to be the number in (1).

The purpose of rendering a double v, though, is *not* to expose every digit of its full decimal expansion, as in (1). Quite the opposite: for "ergonomic" reasons we aim at *the shortest decimal* d_v that still rounds back to v. As 1.2 and the decimal in (1) both round to this same double, the clear winner is $d_v = 1.2$.

1.1 Outline

Here's the outline of the writing.

- It starts with a clear and unambiguous definition of d_v, the shortest decimal that rounds to v, in §3 and some preparatory material in §4–7.
- §8 presents the *non-iterative Schubfach algorithm* to determine d_v.
- A naive implementation of Schubfach requires expensive full precision arithmetic. §9 discusses a result-identical *cheaper, high performance variant* which makes use of approximate, limited precision arithmetic.
- §9.9 translates its most convoluted fragment into Java.
- §10 briefly discusses how to extract the individual digits of d_v for string formatting purposes *without divisions at all*.

While some portions are focused on Java, many results hold independently. The validity of the discussion applies to floats as well (appendix).

2 Notations and conventions

Please read this, even cursorily, and come back here when in doubt.

Names in sans serif font are Java related. Generalizing to other programming environments should be easy, but care is advised.

All over, §n, Rn and Fn refer to section, result and figure n, respectively. Further, the following notations and conventions are used from now on:

1. *fp* is a placeholder for a binary floating-point format, e.g., double or float.
2. $0 \in \mathbb{N}$, $\mathbb{N}^* = \mathbb{N} \setminus \{0\}$.
3. $b, d, f, m, n \in \mathbb{N}$, $e, h, i, j, p \in \mathbb{Z}$, α, $\beta, \gamma \in \mathbb{R}$, x, y denote D-aries (§6).
4. $\lfloor\alpha\rfloor = \max\{i \mid i \leq \alpha\}$ ("floor"), $\lceil\alpha\rceil = \min\{i \mid i \geq \alpha\}$ ("ceiling").
5. When $\beta \neq 0$, $\alpha \mathbin{/\!\!/} \beta = \lfloor\alpha/\beta\rfloor$ ("div"), $\alpha \mathbin{\backslash\!\!\backslash} \beta = \alpha - (\alpha \mathbin{/\!\!/} \beta)\beta$ ("mod").
6. $\alpha \mid \beta$ iff $\beta = i\alpha$ for some i ("α divides β"), $\alpha \nmid \beta$ iff $\neg(\alpha \mid \beta)$.
7. $b \mathbin{[\!]} d$, the *bitwise or operation* on b and d.
8. Some names are reserved:
   (a) v, a *finite positive fp* (§3).
   (b) d_v, the D-ary selected to be formatted as string (§3).
   (c) $c \in \mathbb{N}^*$ and $q \in \mathbb{Z}$ (§4).
   (d) Interval R_v and $c_\ell, c_r, v_\ell, v_r \in \mathbb{R}$ (§5).
   (e) Integer $D \geq 2$, $D \neq 3 \cdot 2^n$ for all n and sets D_i (§6).
   (f) Integers s_i and t_i and D-aries u_i and w_i (§6).
   (g) R_i, a restricted set of D-aries (§7).
   (h) $k \in \mathbb{Z}$ (§8).
   (i) $s, t, s', t' \in \mathbb{N}$ and D-aries u, w, u', w' (§8)

Other names are introduced as needs arise.

3 Rendering

Since the zeroes, the infinities and the NaNs are trivial to render and since rendering a negative *fp* v is substantially the same as rendering a positive *fp*, only finite positive *fp*s are discussed from now on.

Java *fp*s are modeled according to the IEEE 754 specification. A (finite positive) *fp* value v is represented as a *binary* number of the form $v = c2^q$, for some c and q (see §4 for details and item 8c in §2).

When presenting v to a human reader, however, it is *represented by a decimal approximation*, a number of the form $d10^i$, for some d and i (§6). This decimal, acting as a deputy for the original v, is then formatted as a string according to some notation (plain, scientific, etc.), some locale (characters for the digits, for the grouping separator, etc.) and possibly other rules.

Rendering a finite positive *fp* v thus proceeds in two stages:

- A positive decimal d_v is *selected* to represent v.
- The selected d_v is *formatted* as a string.

Given finite positive *fp* v, the D-ary d_v is defined as follows:

- Let *round* be the map used to convert a real number to the closest *fp*.
- $R = \{x \mid round(x) = v\}$ contains all D-aries that round to v.
- $m = \min\{\operatorname{len} x \mid x \in R\}$ is the minimal length of the D-aries in R.
- $T = \{x \in R \mid \operatorname{len} x = m\}$ contains all D-aries in R of length m.
- Define d_v to be the D-ary in T closest to v. Or if there are two such D-aries, let d_v be the one with the even significand.

Figure 1: Definition of d_v: optimal variant

3.1 Most decimals aren't fps

On many platforms, including Java, a decimal, whether obtained from source code, user input or other, undergoes a conversion by rounding it to the closest *fp*. However, rounding means that even simple, low precision decimals like 0.1 and 1.2, are *not* expressed exactly as *fp*s.

Indeed, going back to the example in §1, suppose $1.2 = c2^q$ for some c and q. The right-hand side is an integer when $q \geq 0$, so cannot be 1.2. Thus assume $q < 0$: rewriting leads to $12 \cdot 2^{-q} = 10c$, where both sides are integers. Prime 5 divides the right but not the left-hand side: this cannot be. There are no c and q of *any* magnitude with $1.2 = c2^q$: all the more so, 1.2 can't be a *fp*.

This line of reasoning is quite general and applies to most decimals. It depends on the fact that not every prime dividing 10, the radix of the decimals, also divides 2, the radix of the *fp*s (as prime 5 shows).

3.2 Definition of d_v: optimal variant

Conversely, every prime dividing radix 2 (namely prime 2) also divides radix 10. Consequently, every *fp* $v = c2^q$ is a decimal. In fact, when $q < 0$ then $v = c2^q = c(10^q/5^q) = (c5^{-q})10^q$ has the form of a decimal: both $c5^{-q}$ and q are integers. And when $q \geq 0$ then $v = (c2^q)10^0$ has the form of a decimal as well.

The reader might thus be tempted to directly format the decimal expansion of v. The temptation must be resisted, however, because the expansion is very often quite long and hardly readable, as illustrated in the introduction. (See for example §4.9.1 in [MBD+2018] for a discussion of worst expansion lengths.) Since any D-ary that rounds to v is as good as v itself for the purpose of rendering, d_v is selected to be the *shortest*.

What does this mean? Anticipating the details of §6, every positive decimal x has a unique form $x = d \times 10^i$ where $10 \nmid d$ (meaning that d has no trailing zeroes). The integer d is called the *significand* of the decimal x. The *length* of x, denoted $\operatorname{len} x$, is the number of digits in the decimal expansion of d. For example, $\operatorname{len} 12\,000 = \operatorname{len}(12 \times 10^3) = 2$.

It's rather straightforward to consider a generic radix D, not necessarily 10, and the corresponding generic length (§6). This leads to the optimal variant of the definition of d_v sketched in F1 (see just below for expectations on *round*).

3.2.1 Assumptions about *round*

It is assumed that *round* in F1 maps each real number to the *closest* *fp*, breaking ties in some specific way. For the case of Java, $round = roundTiesToEven$, where

Given finite positive *fp* v and $M \geq 1$, the D-ary d_v is defined as follows:

- Let *round* be the map used to convert a real number to the closest *fp*.
- $R = \{x \mid round(x) = v\}$ contains all D-aries that round to v.
- $m = \min\{\operatorname{len} x \mid x \in R\}$ is the minimal length of the D-aries in R.
- When $m \geq M$, let $T = \{x \in R \mid \operatorname{len} x = m\}$ contain all D-aries in R of length m.
  Otherwise, let $T = \{x \in R \mid \operatorname{len} x \leq M\}$ contain all D-aries in R of length up to M included.
- Define d_v to be the D-ary in T closest to v. Or if there are two such D-aries, let d_v be the one with the even significand.

Figure 2: Definition of d_v: general variant

ties favor the "even" *fp*, the one whose least significant bit is 0.

3.3 Definition of d_v: general variant

The specification of the toString methods in classes Float and Double from the Java standard library dictates that small values must be rendered in *computerized scientific notation*, as in 1.23E-21. The decimal point is always present, there's a non-zero digit to its left and at least another digit to its right.

This requirement causes a cosmetic issue with the definition of F1.

Example 1. Let double $v = 20 \cdot 2^{-1074} = 9.88\ldots \cdot 10^{-323}$, whose full decimal expansion has the unwieldy length of 750, really! The definition determines

$$R = \{1 \cdot 10^{-322},\ 97 \cdot 10^{-324},\ 98 \cdot 10^{-324},\ 99 \cdot 10^{-324}, \ldots \text{(longer decimals)}\}$$
$$m = 1, \qquad T = \{1 \cdot 10^{-322}\}$$

and selects $d_v = 1 \times 10^{-322}$ as the only element in T, which is formatted as 1.0E-322. Since there's room for 2 digits anyway, a better choice would be 99×10^{-324}. Although longer in the sense of §3.2, it is closer to v (and closer than 98×10^{-324}, the other close decimal of length 2), yet still requiring only two digits when formatted: 9.9E-323. Closer, albeit longer decimals of length up to 2 are preferred over shorter but farther ones.

Example 2. The smallest double is $v = 1 \cdot 2^{-1074} = 4.94\ldots \cdot 10^{-324}$, a decimal of length 751. The definition of F1 determines

$$R = \{25 \cdot 10^{-325},\ 26 \cdot 10^{-325},\ \ldots,\ 74 \cdot 10^{-325}, \ldots \text{(longer decimals)}\}$$
$$m = 1, \qquad T = \{3 \cdot 10^{-324}, \ldots, 7 \cdot 10^{-324}\}$$

and selects $d_v = 5 \times 10^{-324}$ from T because it is the one closest to v, formatted as 5.0E-324. The ideal choice when two digit positions are available anyway, though, is 49×10^{-325} because it is even closer to v and formatted as 4.9E-324.

More generally, given $M \in \mathbb{N}^*$, the requirements for d_v are as before but favoring decimals closer to v as long as their length doesn't exceed M. The variant discussed above implicitly has $M = 1$. The general definition is in F2. The only difference lies in what to retain in T when $m < M$.

To fulfill the specification of the toString methods, set $M = 2$ in F2.

For the case of example 1, this variant determines

$$m = 1, \qquad T = \{1 \cdot 10^{-322},\ 97 \cdot 10^{-324},\ 98 \cdot 10^{-324},\ 99 \cdot 10^{-324}\}$$

containing decimals of length 1 or 2. It defines $d_v = 99 \cdot 10^{-324}$, as desired.

With respect to example 2, the definition of F2 determines

$$m = 1, \qquad T = \{25 \cdot 10^{-325},\ 26 \cdot 10^{-325},\ \ldots,\ 74 \cdot 10^{-325}\}$$

T contains 50 decimals of length 1 or 2, yielding $d_v = 49 \cdot 10^{-325}$ as expected.

3.4 Round trips

It is clear that the round trip from a *fp* v to d_v and back to *fp* is the identity, fully recovering v. In fact, this holds for any value chosen from R, but d_v is "best" in the sense of being the shortest (or among the shortest).

What about the opposite round trip, from a decimal x to the closest *fp* v and back to the decimal d_v? Of course, since there are infinitely many decimals that round to the same *fp*, in general the round trip is not the identity. However, when $\operatorname{len} x \leq G$ (§8.1 and F3) and v is normal (§4), it can be shown that, in fact, $d_v = x$. (The full story is more involved, but this fact should prove sufficiently interesting anyway. See [Mat1968] for the details.)

For example, "reading" the decimal $x =$ 11.9999999999999e-1 produces the closest double v. Since $\operatorname{len} x = 15 \leq G$ and since v is normal, it can be anticipated with certainty, despite not knowing v, that F1 and F2 both select $d_v = 119\,999\,999\,999\,999 \times 10^{-14}$, (later formatted as 1.19999999999999) because d_v *must* be x.

4 Finite positive fps

The space of (finite positive) *fp*s is characterized by the following parameters (see also IEEE 754 for specific formats and F3 for some examples):

- the precision P
- the overall size S

They determine

- the exponent width $W = (S - 1) - (P - 1) = S - P$
- the minimal exponent $Q_{\min} = -(2^{W-1}) - P + 3$
- the maximal exponent $Q_{\max} = 2^{W-1} - P$

In this space, each *fp* v has a unique form $v = c \otimes 2^q$, where c and q meet

$$Q_{\min} \leq q \leq Q_{\max}$$

and

$$\begin{array}{lll} \text{either} & 2^{P-1} \leq c < 2^P & \text{(normal)} \\ \text{or} & 0 < c < 2^{P-1} \ \wedge\ q = Q_{\min} & \text{(subnormal)} \end{array}$$

The operator $\otimes$ is used to emphasize that c and q meet these inequalities.

S	P	W	$Q_{\min}$	$Q_{\max}$	$K_{\min}$	$K_{\max}$	G	H	C_{tiny}
16	11	5	−24	5	−8	1	3	5	2
32	24	8	−149	104	−45	31	6	9	8
64	53	11	−1 074	971	−324	292	15	17	3
128	113	15	−16 494	16 271	−4 966	4 898	33	36	2
256	237	19	−262 378	261 907	−78 984	78 841	71	73	5

Figure 3: Parameters from Table 3.5 of IEEE 754 (right-hand half has $D = 10$)

Some extreme values are

$$\begin{aligned}
\mathsf{MIN_VALUE} &= \quad 1 \otimes 2^{Q_{\min}} = 2^{Q_{\min}} \\
\mathsf{MIN_NORMAL} &= \quad 2^{P-1} \otimes 2^{Q_{\min}} = 2^{P-1+Q_{\min}} \\
\mathsf{MAX_VALUE} &= (2^P - 1) \otimes 2^{Q_{\max}}
\end{aligned}$$

5 The rounding interval R_v

The rounding interval R_v associated to $v = c \otimes 2^q$ contains exactly all reals that are closer to v than to any other *fp*, with ambiguities resolved according to some specific tie-breaking rule. The interval spans between its *left endpoint* v_ℓ and its *right endpoint* v_r, and has length $\|R_v\| = v_r - v_\ell$.

More precisely, $v_\ell = c_\ell 2^q$ lies halfway between v and its predecessor, where

$$c_\ell = \begin{cases} c - 1/2, & \text{if } c > 2^{P-1} \vee q = Q_{\min} \quad \text{(regular spacing)} \\ c - 1/4, & \text{otherwise} \quad \text{(irregular spacing)} \end{cases} \tag{2}$$

As alluded, the distinction is due to the irregular spacing between three adjacent *fp*s when the middle v happens to be a power of 2 greater than MIN_NORMAL.

Similarly, $v_r = c_r 2^q$ lies halfway between v and its successor, where

$$c_r = c + 1/2 \tag{3}$$

(IEEE 754 defines the successor of MAX_VALUE to be $+\infty$. Here, however, it is quite naturally the number $2^{P+Q_{\max}}$.)

Note that $c_\ell < c < c_r < 2^P$ and that *irregular spacing implies a normal* v.

There are different tie-breaking rules in use, depending on which of the endpoints are included in R_v. The default IEEE 754 rounding is *roundTiesToEven*: both endpoints are included in R_v if c is even and both are excluded otherwise. Thus, for *roundTiesToEven*

$$R_v = \begin{cases} [v_\ell, v_r], & \text{if } c \text{ is even} \quad \text{(endpoints included)} \\ (v_\ell, v_r), & \text{otherwise} \quad \text{(endpoints excluded)} \end{cases}$$

Other common roundings are *roundTiesToAway*, which breaks ties in favor of greater magnitudes, and *roundTiesToZero*, which breaks them favoring lesser magnitudes. That is, $R_v = [v_\ell, v_r)$ and $R_v = (v_\ell, v_r]$, respectively.

An immediate result is:

Result 1. *Regardless of whether the boundaries, independently from each other, belong to* R_v *or not,* $x \in R_v$ *yields* $v_\ell \leq x \leq v_r$ *and* $v_\ell < x < v_r$ *yields* $x \in R_v$.

6 D-aries

In this writing, a decimal is a non-negative number of the form $d10^i$. However, most of the discussion is valid assuming a more general radix D. For reasons that will become clear in §7, however, D is assumed *not* to have the form $3 \cdot 2^n$ (that is, $D \neq 3, 6, 12, \ldots$). The usual case $D = 10$ fits this assumption.

Definition 1. For any i, let $D_i = \{dD^i\} = D^i\mathbb{N}$ denote the set of all numbers of the indicated form. Such a number is called a *D-ary* (subject to exceptions, like decimal, octal, etc.)

The distance between two adjacent D-aries in D_i is D^i. Thus, for $x, y \in D_i$ (item 3 in §2), the inequality $x < y$ is equivalent to $x + D^i \leq y$.

When does dD^i also belong to D_j? If $dD^i \in D_j$ then $dD^i = bD^j$ for some b, so $d = bD^{j-i}$ which means $D^{j-i} \mid d$. Conversely, if $D^{j-i} \mid d$ then $dD^i = (d/D^{j-i})D^j \in D_j$ because $d/D^{j-i} \in \mathbb{Z}$. Note that item 6 of §2 allows the case $i > j$, so $D^{j-i} \mid d$ holds because $d = (dD^{i-j})D^{j-i}$ and $dD^{i-j} \in \mathbb{Z}$.

Result 2. *$dD^i \in D_j$ iff $D^{j-i} \mid d$. In particular, $i > j$ yields $D_i \subset D_j$.*

6.1 Closest estimates in D_i

Two D-aries in D_i stand out quite naturally as estimates for *fp* v.

Definition 2. Given v, let

$$s_i(v) = \lfloor vD^{-i} \rfloor \qquad t_i(v) = s_i(v) + 1 \qquad u_i(v) = s_i(v)D^i \qquad w_i(v) = t_i(v)D^i$$

When v is clear from the context, we write s_i for $s_i(v)$, etc.

It follows at once that $u_i, w_i \in D_i$ and that $s_i \leq vD^{-i} < t_i$ and therefore $u_i \leq v < w_i$. Because u_i and w_i are adjacent in D_i, for any other $x \in D_i$ either $x < u_i$ or $x > w_i$ holds. In other words:

Result 3. *In D_i, the closest underestimate of v is u_i, and the closest strict overestimate of v is w_i.*

The definition gives rise to $s_j \leq vD^{-j} = vD^{-i}D^{i-j}$, so $s_jD^{j-i} \leq vD^{-i} < t_i$. When $j \geq i$, the left-hand side is an integer, hence $s_jD^{j-i} \leq t_i - 1 = s_i$. From here, it follows that $s_jD^j \leq s_iD^i$, meaning that $u_j \leq u_i$. Similarly, $t_jD^{j-i} \geq t_i$ and $w_j \geq w_i$. We then get $s_j \leq s_i/D^{j-i} < t_j = s_j + 1$, that is, $s_j = s_i /\!\!/ D^{j-i}$.

Result 4. *When $j \geq i$ then*

$$s_jD^{j-i} \leq s_i < t_i \leq t_jD^{j-i} \qquad\qquad u_j \leq u_i < w_i \leq w_j \qquad\qquad s_j = s_i /\!\!/ D^{j-i}$$

6.2 The length of a D-ary

For each $i \geq j$, the equality $x = dD^i = (dD^{i-j})D^j \in D_j$ shows that a D-ary has infinitely many forms. Provided $x > 0$, exactly one of these forms has the smallest d, a fact that is almost obvious and taken for granted.

Definition 3. Let $x > 0$.

- The *shortest form* of $x = d \times D^i$ has $D \nmid d$. Whenever $\times$ appears in a form, as here, it indicates its shortest variant.

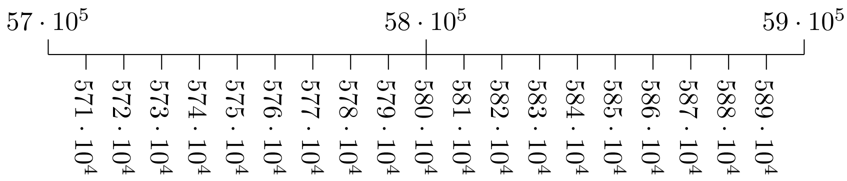

Figure 4: $d = 58$, $i = 5$, $j = 4$: never shorter and usually longer

- The integer d appearing in the shortest form is called the *significand* of x.
- The *length* $\operatorname{len} x$ is the unique n meeting $D^{n-1} \leq d < D^n$, where d is the significand of x. (This is the number of digits in the conventional D-ary expansion of d.)
- y is *shorter* than x iff $\operatorname{len} y < \operatorname{len} x$. Similarly for *longer*, etc.

For example, the significand and length of the decimal $3\,400 \cdot 10^{-3} = 34 \times 10^{-1}$ are 34 and 2, resp. Also, despite the appearance, 12 000 is shorter than 1.23 and is as long as 0.450. Indeed, $12\,000 = 12 \times 10^3$, $1.23 = 123 \times 10^{-2}$ and $0.450 = 45 \times 10^{-2}$, with lengths 2, 3 and 2, resp.

6.3 Local properties of the length

The granularity of D_{i-1} is D times finer than that of D_i, so its elements are more precise. In the proximity of any non-zero element of D_i, therefore, the elements of D_{i-1} are expected never to be shorter and often to be longer.

To make this intuition precise, let $x = dD^i > 0$ (any form does). Further, let $I = (x - D^i, x + D^i)$ and note that $D_i \cap I = \{x\}$. Given j, assume $y \in D_j \cap I$, so $y = b \times D^h$ for some b and $h \geq j$.

- When $h \geq i$ (in particular, when $j \geq i$), then $y \in D_h \subseteq D_i$ (R2), so $y \in D_i \cap I$, thus $y = x$ and $\operatorname{len} y = \operatorname{len} x$. Also, $y \neq x$ yields $h < i$.
- Otherwise $h < i$. Let $\ell = i - h$, so $\ell > 0$, and note that, since $D \nmid b$, it means $b \neq dD^\ell$, yielding $bD^h \neq dD^i$, so $y \neq x$.
  Let $n = \operatorname{len} x$, hence $d \geq D^{n-1}$. Note that $d = D^{n-1}$ holds iff $n = d = 1$.
  - If $y > x$ then $b > dD^\ell$, thus $b > D^{n-1}D^\ell \geq D^n$, giving $\operatorname{len} y > \operatorname{len} x$.
  - If $y < x$ and $d > 1$ then, as $y \in I$, it follows $y > x - D^i = (d-1)D^i$: that is, $b > (d-1)D^\ell$. Furthermore, $d > 1$ means $d > D^{n-1}$, hence $d - 1 \geq D^{n-1}$, so $b > D^{n-1}D^\ell \geq D^n$, implying $\operatorname{len} y > \operatorname{len} x$.
  - Otherwise $y < x$ and $d = 1$, so, obviously, $\operatorname{len} y \geq 1 = \operatorname{len} x$.

This proves (see F4 and F5)

Result 5. *Let $x = dD^i > 0$ and $y \in D_j \cap (x - D^i, x + D^i)$. Then $\operatorname{len} y \geq \operatorname{len} x$. Further, $j \geq i$ implies $y = x$ and $y > x \vee (y < x \wedge d > 1)$ yields $\operatorname{len} y > \operatorname{len} x$.*

Assume $n \in \mathbb{N}^*$ and $d \geq D^{n-1}$ and let $x = dD^i$ and $y \in (x, x + D^i)$. At least one of $D \nmid d$ or $D \nmid d+1$ must hold. Since $d + 1 > d \geq D^{n-1}$ this means that at least one of $\operatorname{len} x \geq n$ or $\operatorname{len}(x + D^i) \geq n$ is certainly met. But R5 clearly shows $\operatorname{len} y > \operatorname{len} x$. Since $d + 1 > 1$, it also shows $\operatorname{len} y > \operatorname{len}(x + D^i)$.

Result 6. *Let $n \in \mathbb{N}^*$, $d \geq D^{n-1}$, $x = dD^i$ and $y \in (x, x + D^i)$. Then*

$$\operatorname{len} y > \max(\operatorname{len} x, \operatorname{len}(x + D^i)) \geq n$$

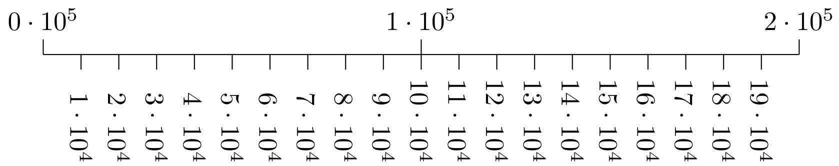

Figure 5: $d = 1$, $i = 5$, $j = 4$: never shorter and usually, *but not always*, longer

Now suppose $n \in \mathbb{N}^*$, $D^n \le d_\ell < d_r$ with $d_\ell, d_r \in \mathbb{N}$ and $D \nmid d$ for each $d \in [d_\ell, d_r]$. Also let $m = \operatorname{len}(d_\ell D^i)$. This means that $D^{m-1} \le d_\ell < D^m$, so $d_\ell + 1 < D^m$ as well, since $D \nmid d_\ell + 1$. Therefore, $m = \operatorname{len}((d_\ell + 1)D^i)$, too. Continuing this way, we get $\operatorname{len}(dD^i) = m$ for each $d \in [d_\ell, d_r]$. In addition, $m > n$ because $d_\ell \ge D^n$. Combining with the last result gives

Result 7. *Let $n \in \mathbb{N}^*$, $D^n \le d_\ell \le d_r$, with $d_\ell, d_r \in \mathbb{N}$ and $D \nmid d$ for each $d \in [d_\ell, d_r]$.*

- $\operatorname{len}(dD^i) = \operatorname{len}(d'D^i)$ *for each* $d, d' \in [d_\ell, d_r]$. *Further, this common length m meets $m > n$.*
- *Let $y \in ((d_\ell - 1)D^i, (d_r + 1)D^i) \setminus D_i$. Then* $\operatorname{len} y > m$.

7 Fitting D-aries into R_v

Consider both the rounding interval R_v of $v = c \otimes 2^q$ and any grid-like set D_i. Central to Schubfach is the question of when D_i and R_v intersect. To this end, it is essential to compare the distance D^i between adjacent D-aries in D_i with the width $\|R_v\|$ of the rounding interval.

Definition 4. Let $R_i(v) = D_i \cap R_v$.

When v is clear from the context, we write R_i for $R_i(v)$.

When $D^i > \|R_v\|$, it should be clear that R_i contains at most one element. Indeed, suppose $x, y \in R_i$. Consequently, $|x - y| = nD^i$ for some n (since $x, y \in D^i$) and $|x - y| \le \|R_v\| < D^i$ (since $x, y \in R_v$), so $n = 0$ and $x = y$.

Result 8. *When $D^i > \|R_v\|$, there's at most one $x \in R_i$ (top of F6).*

When $D^i < \|R_v\|$ then, intuitively, R_i contains at least one element. To show this, consider $x = \max\{y \in D_i \mid y < v_r\}$. The definition of x leads to $x < v_r \le x + D^i < x + \|R_v\|$, thus $v_\ell = v_r - \|R_v\| < x$. This gives $v_\ell < x < v_r$ and, by R1, $x \in R_v$, so $R_i \ne \varnothing$.

When $D^i = \|R_v\|$ it is necessary to consider the spacing around v.

- Regular spacing means $\|R_v\| = 2^q$ by (2) and (3). Therefore, $v = c2^q = cD^i \in D_i \cap R_v$: that is, $R_i \ne \varnothing$.
- Otherwise spacing is irregular: $\|R_v\| = 3/4 \cdot 2^q$. But then $D^i = \|R_v\|$ is equivalent to $4D^i = 3 \cdot 2^q$, which never holds when $D \ne 3 \cdot 2^n$, as assumed in §6. That is, $D^i = \|R_v\|$ is never met when spacing is irregular.
- Therefore, when $D^i = \|R_v\|$ then $R_i \ne \varnothing$.

Result 9. *When $D^i \le \|R_v\|$, there's at least one $x \in R_i$ (bottom of F6).*

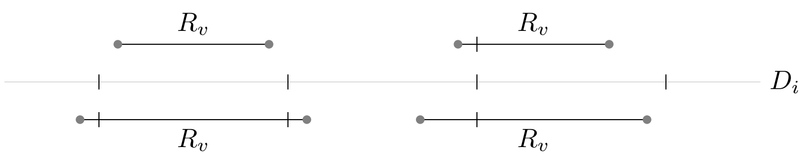

Figure 6: Middle: ticks denote adjacent D_i elements. Regardless of the position of R_v w.r.t. D_i, according to R8 and R9, resp., catching more than one tick (top) and avoiding any tick (bottom) goes badly. Ticks on R_v constitute R_i

8 The Schubfach algorithm

The results in §7 are based on comparing the distance of adjacent values in some D_i to the width of the rounding interval R_v. The underlying idea is related to the well-known *pigeonhole principle* first formalized by the great mathematician J. P. G. Lejeune Dirichlet under the German name of *Schubfachprinzip*. The algorithm presented here builds on it, thus its name which deliberately departs from a long lineage of fabulous drakes.

R8 and R9 are combined in

Result 10. *Let k be the unique integer meeting $D^k \leq \|R_v\| < D^{k+1}$, that is, $k = \lfloor \log_D \|R_v\| \rfloor$. R_k contains* at least one *and R_{k+1}* at most one *element.*

Hence, k sets apart those R_i that can never be empty from those that could. Schubfach usually only deals with values in R_k and R_{k+1}. The reason is that values in R_i are never shorter and usually longer than those in R_k when $i < k$ while they are all already contained in R_{k+1} when $i > k+1$.

The exception happens when $M = 2$ in F2 and v is so tiny that $s_k < D$, so $\operatorname{len} u_k = \operatorname{len} w_k = 1$. Since it is required to examine values of length 2 as well when they are closer to v (§3.3), R_{k-1} is added for consideration in such cases.

Note that $\|R_{\mathsf{MIN\_VALUE}}\| = 2^{Q_{\min}}$ and $\|R_{\mathsf{MAX\_VALUE}}\| = 2^{Q_{\max}}$. Let's also introduce some names for succinctness:

Definition 5. With k as in R10, let

$$K_{\min} = \lfloor \log_D 2^{Q_{\min}} \rfloor \qquad K_{\max} = \lfloor \log_D 2^{Q_{\max}} \rfloor \qquad k' = k + 1$$

$$s = s_k \qquad t = t_k \qquad u = u_k \qquad w = w_k$$

$$s' = s_{k'} \qquad t' = t_{k'} \qquad u' = u_{k'} \qquad w' = w_{k'}$$

When $x \in R_i$, R3 says that $x \leq v$ yields $u_i \in R_i$ and that $x > v$ yields $w_i \in R_i$. This permits to reformulate R10 as

Result 11. *At least one of $u \in R_k$, $w \in R_k$ holds. Also, either $R_{k'} = \varnothing$, or $R_{k'} = \{u'\}$, or $R_{k'} = \{w'\}$.*

8.1 Useful bounds

The following integers turn out to be useful (see F3 for some values)

Definition 6.

$$G = \max\{j \mid D^j \leq 2^{P-1}\} \qquad H = \min\{i \mid 2^P < D^{i-1}\}$$

(Similar quantities are studied in [Mat1968] about conversions between finite precision spaces. That work, however, is not about shortest outcomes.) Observe that $0 \le G \le H - 2$.

Some useful bounds related to G and H are found as follows.

- First consider irregular spacing, so $\|R_v\| = 3/4 \cdot 2^q$ and $c = 2^{P-1}$. The definition of k entails

$$1 < 4/3 \le 2^q D^{-k} < 4/3 \cdot D < 2D \qquad \text{(irregular spacing)} \tag{4}$$

  Then $s'D \le vD^{-k'}D = c(2^q D^{-k}) < 2^{P-1}(2D) = 2^P D \le (D^H - 1)D$; consequently, $t'D = (s' + 1)D = s'D + D < D^H$.
  Also, $v_r D^{-k} = c_r 2^q D^{-k} < (2^{P-1}+1/2)2D = (2^P+1)D \le D^{H-1}D = D^H$.
- Otherwise $\|R_v\| = 2^q$, so

$$1 \le 2^q D^{-k} < D \qquad \text{(regular spacing)} \tag{5}$$

  Hence, $s'D \le vD^{-k'}D = c(2^q D^{-k}) < cD < 2^P D \le (D^{H-1} - 1)D = D^H - D$; again, $t'D = s'D + D < D^H$.
  Moreover, $v_r D^{-k} = c_r 2^q D^{-k} < 2^P D < D^{H-1}D = D^H$, again.
- We have $t > vD^{-k} = c(2^q D^{-k}) \ge c \ge D^{\lfloor \log_D c \rfloor}$; hence, $s = t - 1 \ge D^{\lfloor \log_D c \rfloor}$. In particular, when v is normal (thus including irregular spacing) it follows that $\lfloor \log_D c \rfloor \ge \lfloor \log_D 2^{P-1} \rfloor = G$.

Result 12. *Recalling that $t \le t'D$ (R4), the notes above show*

$$v_r D^{-k} < D^H$$
$$D^{\lfloor \log_D c \rfloor} \le s < t \le t'D < D^H$$

In particular, $D^G \le s$ for normal v (hence including irregular spacing).

As a consequence, $\operatorname{len} u \le H$ *and* $\operatorname{len} w \le H$.

In the usual case $D = 10$, from F3 we have $D^H = 10^9 < 2^{30}$ for floats *and $D^H = 10^{17} < 2^{57}$ for* doubles.

8.2 Skeleton of Schubfach

F1 and F2 are useful as abstract definitions but are ineffective as algorithms because the set R contains infinitely many decimals.

While R11 shows that at least one of u, $w \in R_v$, the question arises whether shorter values exist as well, which is the aim while selecting d_v (with additional provisions when $M = 2$). For example, if either u' or w' happens to lie in R_v then it is often shorter.

Here's how Schubfach works: M, m and T are from F1 and F2, let $y \in R_v$ and either $M = 1$ (optimal variant) or $M = 2$ (Java's toString).

- First assume $s \ge D^M$. (From R12, as soon as $c \ge D^M$ then $\lfloor \log_D c \rfloor \ge M$, thus $s \ge D^M$; that is, except for tiny values, the assumption holds.)
  - Suppose $R_{k'} \ne \varnothing$, so $R_{k'} = \{x\}$ for either $x = u'$ or $x = w'$, and note that $t' > s' \ge D^{M-1}$ (R4 and $D^{M-1} \in \mathbb{Z}$). Set M, u', s', k' for n, x, d, i, resp., in R6 to obtain $\operatorname{len} u' \ge M$ or $\operatorname{len} w' \ge M$. Also, let $I = (x - D^{k'}, x + D^{k'})$. Its boundaries are well outside R_v (R10), so $R_v \subseteq I$ and $y \in I$.

– Assume $u' \in R_v$ and let $n = \operatorname{len} u'$. When $y > u'$ then $y \in (u', w')$. Hence, $\operatorname{len} y > n$ and $\operatorname{len} y > M$ (set u', s', k' for x, d, i, resp., in R6), so $y \notin T$. When $y < u'$ then y is neither shorter than u' (R5) nor closer to v (R3). It follows that $m = n$, $\max T = u'$ and $d_v = u'$.
– Otherwise $w' \in R_v$. A similar, slightly simpler reasoning leads to $m = \operatorname{len} w'$, $T = \{w'\}$ (a singleton) and $d_v = w'$.
Summary: *When $s \geq D^M$ and either $u' \in R_v$ or $w' \in R_v$, then either $d_v = u'$ or $d_v = w'$, resp.*
– Otherwise $u', w' \notin R_v$, which means $R_{k'} = \varnothing$. Take any $dD^k \in R_k$. Because $R_{k'} = \varnothing$, we have $D \nmid d$ (R2).
– Since $R_k \neq \varnothing$, we have $R_k = \{dD^k \mid d \in \{d_\ell, \ldots, d_r\}\}$, with $d_\ell \leq d_r$ and $d_\ell, d_r \in \mathbb{N}$. Thus $D \nmid d$ for each $d \in \{d_\ell, \ldots, d_r\}$. Let n denote the common length (R7) of any element of R_k.
– Of course, s or t belong to $\{d_\ell, \ldots, d_r\}$ and $t > s \geq D^M$, so $n > M$. Further, when $y \notin R_k$ we have $\operatorname{len} y > n$ (R7).
– Consequently, $m = n > M$ and $T = R_k$. By R3, only u or w is selected to be d_v, the others being farther away from v.
– ***Summary:*** *If $s \geq D^M$ and $u', w' \notin R_v$, then $d_v = u$ or $d_v = w$.*

- Now assume $D \leq s < D^M$. This is the case illustrated by example 1 and is, of course, never met when $M = 1$, thus $M = 2$.
  – From $s < D^M$ we get $\operatorname{len} u, \operatorname{len} w \leq M = 2$. Also, $\operatorname{len} u \geq 2$ or $\operatorname{len} w \geq 2$ (set $2, u, s, k$ for n, x, d, i, resp., in R6), so at least one of u or w has length $2 = M$ and the other has length 1 or 2.
  – If $y < u$ then $u \in R_v$, so $u \in T$. Regardless of $\operatorname{len} y$, u is closer to v, hence $d_v \neq y$.
  – Similarly, $y > w$ leads to $d_v \neq y$.
  – R6 also shows $\operatorname{len} y > M$ for $y \in (u, w)$, so $y \notin T$.
  – ***Summary:*** *When $D \leq s < D^M$ then $d_v = u$ or $d_v = w$.*
- Otherwise $s < D$. Example 2 illustrates this case.
  Here, $\operatorname{len} u = \operatorname{len} w = 1$.
  – Let $M = 1$. As above, any y other than u and w is either longer, so $y \notin T$, or farther from v than u (when $y < u$) or than w (when $y > w$), whereby $d_v \neq y$.
  – ***Summary:*** *When $s < D$ and $M = 1$ then $d_v = u$ or $d_v = w$.*
  – Otherwise $M = 2$. It then might happen that T also includes values of length 2 that are closer to v than u or w. However, knowledge of s alone doesn't give enough clues for these longer values.
  – R4 shows $D \leq s_{k-1} < D^2 = D^M$. A reasoning similar to the one used in case $D \leq s < D^M$ above leads to
  Summary: *When $s < D$ and $M = 2$ then $d_v = u_{k-1}$ or $d_v = w_{k-1}$.*

8.2.1 Dealing with tiny values

As mentioned, Schubfach must deal exceptionally when $s < D$ and $M = 2$. But while knowledge of s is sufficient to determine s' in the case $s \geq D^M$ (R4), it is too coarse for computing s_{k-1}. In a sense, after realizing that $s < D$ holds, it's too late to determine s_{k-1} from s. We'd like earlier detection.

The chain $s < D \Leftrightarrow vD^{-k} < D \Leftrightarrow c2^q D^{-k} < D \Leftrightarrow c < 2^{-q} D^{k+1}$ follows at once from $\lfloor \alpha \rfloor < i \Leftrightarrow \alpha < i$. Inversion of (4) and (5) leads to $2^{-q} D^k \leq 1$, and assuming $G > 0$ we further get $s < D \Leftrightarrow c < 2^{-q} D^{k+1} \leq D \leq D^G \leq 2^{P-1}$. So

if $M = 2 \land c < C_{\text{tiny}}$ then $v \leftarrow vD$, $\Delta k \leftarrow -1$ else $\Delta k \leftarrow 0$ fi
determine k and s
if $s \geq D^M$ then
    $k' \leftarrow k + 1$, $s' = s \mathbin{/\!/} D$, $u' \leftarrow s'D^{k'}$, $w' \leftarrow (s' + 1)D^{k'}$
    if $u' \in R_v$ then return u' fi
    if $w' \in R_v$ then return w' fi
fi
$u \leftarrow sD^k$, $w \leftarrow (s + 1)D^k$
if $u \in R_v \land w \notin R_v$ then return $uD^{\Delta k}$ fi
if $u \notin R_v \land w \in R_v$ then return $wD^{\Delta k}$ fi
if $v - u < w - v$ then return $uD^{\Delta k}$ fi
if $v - u > w - v$ then return $wD^{\Delta k}$ fi
if s is even then return $uD^{\Delta k}$ fi
return $wD^{\Delta k}$

Figure 7: The Schubfach algorithm for $v = c \otimes 2^q$ ($M = 1$ or $M = 2$)

$s < D$ means that $v = c \otimes 2^q$ is subnormal, yielding $q = Q_{\min}$, regular spacing and $k = K_{\min}$. Recalling that $i < \alpha \Leftrightarrow i < \lceil \alpha \rceil$, we obtain (also see F3)

$$C_{\text{tiny}} = \lceil 2^{-Q_{\min}} D^{K_{\min}+1} \rceil$$
$$s < D \quad \Leftrightarrow \quad c < C_{\text{tiny}} \qquad\qquad (G > 0)$$

This cheap test on c allows a precise early detection of the exceptional case. Observe that $s_{k-1} = \lfloor vD^{-(k-1)} \rfloor = \lfloor (vD)D^{-k} \rfloor$ and $u_{k-1} = s_{k-1}D^{k-1} = s_{k-1}D^kD^{-1}$. That is, $s_{k-1}(v) = s_k(vD)$ and that $u_{k-1}(v) = u_k(vD) \cdot D^{-1}$. Tiny values are processed with v replaced by vD and by decrementing the exponent of the outcome by 1.

8.2.2 Pseudocode for Schubfach

The above leads to the Schubfach algorithm of F7.

8.3 A fast path when $q < 0$ and $v \in \mathbb{Z}$

Because $q < 0$ implies $\lfloor \log_D(3/4 \cdot 2^q) \rfloor \leq \lfloor \log_D 2^q \rfloor \leq \lfloor \log_D 2^{-1} \rfloor < 0$, it follows that $k < 0$. Also, $v \in \mathbb{Z}$ entails both $s = \lfloor vD^{-k} \rfloor = vD^{-k}$ and $D \mid s$, and hence $s \geq D$ and $s' = s/D$. Both variants of Schubfach return $u' = u = v = c2^q$.

To test whether $v \in \mathbb{Z}$, observe that $v = c2^q = c/2^{-q} \in \mathbb{Z}$ is equivalent to $c \mathbin{/\!/} 2^{-q} = c/2^{-q}$. This, in turn, holds iff $c = (c \mathbin{/\!/} 2^{-q})2^{-q}$. Further, $q \leq -P$ implies $v = c2^q < 2^P 2^{-P} = 1$, so $v \notin \mathbb{Z}$, showing that $q > -P$ is necessary for $v \in \mathbb{Z}$, which is used as a shortcut to avoid most negative q values in

Result 13. *If* $-P < q < 0$ *and* $c = (c \mathbin{/\!/} 2^{-q})2^{-q}$ *then* $d_v = c \mathbin{/\!/} 2^{-q}$.

The condition on c is nothing else than a right shift, a left shift and a comparison. The condition on q ensures that the shift distance is inside the $[0, 64)$ range for longs when dealing with doubles ($P = 53$, see F3).

9 Efficient computations

A straightforward implementation of Schubfach requires expensive, full precision arithmetic on rational numbers. While it is possible to lower the rationals to integers by multiplication with an appropriate factor, full precision integers (like BigInteger) are still needed. This section focuses on decreasing the costs for arithmetic by using reduced, fixed precision estimates of some crucial quantities.

The plan of the discussion is the following:

- Starting from v, it is almost trivial to extract c and q. This point is not discussed further.
- Compute k as detailed in §9.1.
- Rewrite the tests in Schubfach to involve the fixed precision integers s, t, s' and t' rather than the decimals u, w, u' and w'. To this end, a new set of rationals V, V_ℓ and V_r is introduced (§9.2).
- By applying a specific rounding $r_\mathrm{o} : \mathbb{R} \to \mathbb{Z}$ to V, V_ℓ and V_r, a new set of fixed precision integers $\bar{v}$, $\bar{v}_\ell$ and $\bar{v}_r$ is introduced. Further rewrite the tests to involve these, so that the new tests only involve a few operations on them (§9.3).
- The integer $\bar{v}$ allows a simple computation of s (§9.4).
- Rather than applying the rounding r_o to the rationals V, V_ℓ and V_r directly, which would be cumbersome and expensive, a set of limited precision estimates V', V'_ℓ and V'_r is assumed (but not yet defined) and shown to meet two crucial results (§9.5).
- The rounding r_o is modified to define a slightly different rounding r'_o. The aim is to ensure that r'_o applied to V', V'_ℓ and V'_r has the same outcomes as the original r_o applied to V, V_ℓ and V_r (§9.6).
- The limited precision estimates V', V'_ℓ and V'_r are eventually defined. Their determination involves access to precomputed estimates of powers of D which are held in a lookup table (§9.8).
- Finally, the integers $\bar{v}$, $\bar{v}_\ell$ and $\bar{v}_r$ are computed. The discussion does this by presenting a Java implementation, leading to the apparently obscure code fragment of F8.

9.1 Computing k

After the bits of $v = c \otimes 2^q$ have been used to determine c and q, the next step is the determination of k defined in R10. This entails computing either $\lfloor \log_D 2^q \rfloor$ or $\lfloor \log_D(3/4 \cdot 2^q) \rfloor$. Later, in §9.8.2, the computation of $\lfloor \log_2 D^{-k} \rfloor$ is needed as well.

This section shows efficient computations of $\lfloor \log_D 2^e \rfloor$, $\lfloor \log_D(3/4 \cdot 2^e) \rfloor$ and $\lfloor \log_2 D^e \rfloor$ for a range of exponents sufficiently large for most practical applications, including the computation of k. No iteration is involved.

At first, consider the more involved computation of $m = \lfloor \log_D(3/4 \cdot 2^e) \rfloor = \lfloor e \log_D 2 + \log_D(3/4) \rfloor$. If L and F are good estimates of $\log_D 2$ and $\log_D 3/4$, resp., then $m = \lfloor eL + F \rfloor$ should hold for a sufficiently wide range of e values, despite that neither of the two logarithms is necessarily known exactly.

Given $Q \in \mathbb{N}$, define

$$C = \lfloor 2^Q \log_D 2 \rfloor \qquad L = C 2^{-Q} \qquad A = \lfloor 2^Q \log_D(3/4) \rfloor \qquad F = A 2^{-Q}$$

The larger Q, the better the estimates L and F. With these values,

$$eL + F = (eC + A)2^{-Q}$$

Let ϕ, ψ, ζ, ξ be numbers such that

$$\phi < \log_D 2 < \psi \qquad\qquad \zeta < \log_D(3/4) < \xi$$

We get

$$\lfloor e\phi + \zeta \rfloor \le \lfloor \log_D(3/4 \cdot 2^e) \rfloor \le \lfloor e\psi + \xi \rfloor \qquad (e \ge 0) \tag{6}$$

$$\lfloor e\psi + \zeta \rfloor \le \lfloor \log_D(3/4 \cdot 2^e) \rfloor \le \lfloor e\phi + \xi \rfloor \qquad (e < 0) \tag{7}$$

When the left- and right-hand sides of (6) or (7) are equal to each other and to $\lfloor (eC + A)2^{-Q} \rfloor$, then the latter is equal to the middle side, which is our aim.

A throwaway program using slow but high precision arithmetic and appropriate values for ϕ, ψ, ζ, ξ can pre-compute bounds $e_{\min}$ and $e_{\max}$ for e to meet these conditions. Then, when $e_{\min} \le e \le e_{\max}$, it is certain that $\lfloor (eC + A)2^{-Q} \rfloor$ and $\lfloor \log_D(3/4 \cdot 2^e) \rfloor$ deliver the same results.

9.1.1 Practical values

Determining a good value for Q is a kind of trial-and-error process. Larger values for Q lead to more precise estimates L and F, and thus a larger range for safe e values. On the other hand, a larger range carries higher computational costs for $\mathsf{e * C + A \gg Q} = (eC + A) \mathbin{/\!\!/} 2^Q = \lfloor (eC + A)2^{-Q} \rfloor$. For the case $D = 10$, a satisfactory choice which permits using only `long` arithmetic is

Result 14. *To compute* $\lfloor \log_{10}(3/4 \cdot 2^e) \rfloor$ *let* $Q = 41$. *Then*

$$C = 661\,971\,961\,083 \qquad\qquad A = -274\,743\,187\,321$$
$$e_{\min} = -3\,606\,689 \qquad\qquad e_{\max} = 3\,150\,619$$

so $e_{\min} \le e \le e_{\max}$ *yields* $\lfloor \log_{10}(3/4 \cdot 2^e) \rfloor = \lfloor (eC + A)2^{-Q} \rfloor = \mathsf{e * C + A \gg Q}$

A similar, yet somewhat simpler line of reasoning leads to the following results for the other two computations:

Result 15. *To compute* $\lfloor \log_{10}(2^e) \rfloor$ *let* $Q = 41$. *Then*

$$C = 661\,971\,961\,083 \qquad\qquad e_{\min} = -6\,432\,162 \qquad\qquad e_{\max} = 6\,432\,162$$

so $e_{\min} \le e \le e_{\max}$ *yields* $\lfloor \log_{10}(2^e) \rfloor = \lfloor eC2^{-Q} \rfloor = \mathsf{e * C \gg Q}$

Result 16. *To compute* $\lfloor \log_2(10^e) \rfloor$ *let* $Q = 38$. *Then*

$$C = 913\,124\,641\,741 \qquad\qquad e_{\min} = -1\,838\,394 \qquad\qquad e_{\max} = 1\,838\,394$$

so $e_{\min} \le e \le e_{\max}$ *yields* $\lfloor \log_2(10^e) \rfloor = \lfloor eC2^{-Q} \rfloor = \mathsf{e * C \gg Q}$

These ranges for e should prove sufficient for most practical purposes.

9.2 Computations for Schubfach

As anticipated, here's a new set of variables:

Definition 7. Given v and k as in R10, let

$$V_\ell = v_\ell D^{-k} \qquad V = vD^{-k} \qquad V_r = v_r D^{-k}$$

R12 immediately yields

Result 17. *We have* $V_\ell < V < V_r < D^H$.

Once k has been computed as in §9.1, it is possible to proceed by determining the values from definition 2. What needs to be computed? Surely, $s = \lfloor V \rfloor$ is the starting point, but the discussion on computing it is postponed a bit. Further, Schubfach depends on finding out whether u, w, u', w' belong to R_v. Also, a test to determine which of u or w is closer to v is needed sometimes.

- The test $u \in R_v$ can be rewritten as $v_\ell \preceq_\ell u \preceq_r v_r$, where $\preceq_\ell$ denotes either $\leq$ (when $v_\ell \in R_v$) or $<$ (when $v_\ell \notin R_v$) and analogously for $\preceq_r$. But $u \leq v < v_r$ holds, so $u \preceq_r v_r$ always holds and $u \in R_v$ is simply $v_\ell \preceq_\ell u$ which is equivalent to $V_\ell \preceq_\ell s$.
- Similarly, the test $w \in R_v$ has the same outcome as $t \preceq_r V_r$.
- The tests $u' \in R_v$ and $w' \in R_v$ are carried out analogously.
- To determine which of u or w is closer to v, when need arises, means the same as determining the outcome of $v - u \lesseqgtr w - v$ or, equivalently, that of $2v \lesseqgtr u + w$, which can be rewritten as $2V \lesseqgtr s + t$.

$$
\begin{array}{cccccc}
u \in R_v & \Leftrightarrow & V_\ell \preceq_\ell s & w \in R_v & \Leftrightarrow & t \preceq_r V_r \\
u' \in R_v & \Leftrightarrow & V_\ell \preceq_\ell s'D & w' \in R_v & \Leftrightarrow & t'D \preceq_r V_r
\end{array}
$$

$$v - u \lesseqgtr w - v \quad \Leftrightarrow \quad 2V \lesseqgtr s + t$$

$$s'D = s - s \setminus\!\!\setminus D \qquad t'D = s'D + D$$

9.3 Rounding to odd

Each of the rewritten test above involves a fixed precision integer on one side of the inequality but also a rational value on the opposite side.

By using fixed precision integer estimates for V_ℓ, V_r and V, the aim is to have the same results with more efficient arithmetic. Testing $2V \lesseqgtr s + t$, for example, shall produce the same result as testing $2l \lesseqgtr s + t$, where integer l is an estimate of V, and similarly for the other tests.

The estimates are determined by a rounding $\mathbb{R} \to \mathbb{Z}$, $x \mapsto l$ that can ensure $x \lesseqgtr h \Leftrightarrow l \lesseqgtr h$. Here's a promising attempt:

Definition 8.

$$r_\mathrm{o}(x) = \begin{cases} 2\lfloor x/2 \rfloor, & \text{if } x/2 = \lfloor x/2 \rfloor \\ 2\lfloor x/2 \rfloor + 1, & \text{otherwise} \end{cases}$$

A first property follows almost immediately.

- When x is an even integer, then $x = 2h$, hence the upper branch is activated and $r_\mathrm{o}(x) = x$.

- Otherwise x lies between two adjacent even integers: $2h < x < 2(h+1)$, so $2h+1$ is the odd integer closest to x. The lower branch is activated and $h < x/2 < h+1$, so $\lfloor x/2 \rfloor = h$ and $2\lfloor x/2 \rfloor + 1 = 2h+1$.

This shows that r_o rounds x to the closest *odd* integer (whence the subscript), except when x is an even integer, in which case it rounds it to itself. (A similar rounding for general floating-point arithmetic is discussed in [BM2005].)

This rounding has the crucial property that $x \lesseqgtr 2h \Leftrightarrow r_\mathrm{o}(x) \lesseqgtr 2h$.

- This is clear when x is an even integer, as then $r_\mathrm{o}(x) = x$.
- Otherwise x is not an even integer.
  - When $x > 2h$, then $\lfloor x/2 \rfloor \geq h$, so $r_\mathrm{o}(x) = 2\lfloor x/2 \rfloor + 1 > 2h$.
  - Otherwise $x < 2h$; then $\lfloor x/2 \rfloor \leq h-1$, hence $r_\mathrm{o}(x) = 2\lfloor x/2 \rfloor + 1 \leq 2h - 1 < 2h$.

This is not quite what is needed, though, because s', t', s, t and $s+t$ appearing in the tests above, while integers, are not guaranteed to be even.

One way out would be to multiply everything by 2. Instead, the first four tests are multiplied by 4 and only the last one by 2. Each of V, V_ℓ and V_r is thus multiplied by 4, simplifying the code.

In addition, by using integer estimates the first four tests end up having the form $i \preceq j$. But such an inequality has the same outcome as $i + \mathit{out} \leq j$, where $\mathit{out} = 1$ when $\preceq$ stands for $<$ and $\mathit{out} = 0$ otherwise. Below, *lout* and *rout* play similar roles.

Result 18. *Let*

$$\mathit{lout} = \begin{cases} 0, & \text{if } v_\ell \in R_v \\ 1, & \text{otherwise} \end{cases} \qquad \mathit{rout} = \begin{cases} 0, & \text{if } v_r \in R_v \\ 1, & \text{otherwise} \end{cases}$$

and

$$\bar{v}_\ell = r_\mathrm{o}(4V_\ell) \qquad \bar{v} = r_\mathrm{o}(4V) \qquad \bar{v}_r = r_\mathrm{o}(4V_r)$$

Then

$$\begin{aligned} u' \in R_v &\Leftrightarrow \bar{v}_\ell + \mathit{lout} \leq 4(s'D) & w' \in R_v &\Leftrightarrow 4(t'D) + \mathit{rout} \leq \bar{v}_r \\ u \in R_v &\Leftrightarrow \bar{v}_\ell + \mathit{lout} \leq 4s & w \in R_v &\Leftrightarrow 4t + \mathit{rout} \leq \bar{v}_r \end{aligned}$$
$$v - u \lesseqgtr w - v \quad \Leftrightarrow \quad \bar{v} \lesseqgtr 2(s+t)$$

For doubles *and $D = 10$, every value on the right-hand side of $\Leftrightarrow$ is an integer comfortably fitting in a* long *(R12 and R17).*

9.4 Computing s

Before applying the tests, $s = \lfloor V \rfloor$ must be computed, which can be carried out directly on $\bar{v}$. Given that $\bar{v} = r_\mathrm{o}(4V)$ is approximately $4V$ and that only the floor of V is needed, it might be worthwhile to give a try to $\lfloor \bar{v}/4 \rfloor = \bar{v} \mathbin{/\!\!/} 4$.

- The upper branch of r_o gives rise to $r_\mathrm{o}(4V)/4 = 2\lfloor 2V \rfloor/4 = \lfloor 2V \rfloor/2 = 2V/2 = V$. This means that $\lfloor r_\mathrm{o}(4V)/4 \rfloor = \lfloor V \rfloor = s$.
- The lower branch implies $r_\mathrm{o}(4V)/4 = (2\lfloor 2V \rfloor + 1)/4 = (\lfloor 2V \rfloor + 1/2)/2$. With R28, $r_\mathrm{o}(4V) \mathbin{/\!\!/} 4 = (\lfloor 2V \rfloor + 1/2) \mathbin{/\!\!/} 2 = \lfloor 2V \rfloor \mathbin{/\!\!/} 2 = \lfloor 2V/2 \rfloor = \lfloor V \rfloor = s$.

Result 19. $s = \bar{v} \mathbin{/\!\!/} 4$

9.5 Using estimates for V, V_ℓ and V_r

The tests are now between integers. The evaluations of $\bar{v} = r_o(4V)$, $\bar{v}_\ell = r_o(4V_\ell)$ and $\bar{v}_r = r_o(4V_r)$, however, still involve the cumbersome rationals V, V_ℓ and V_r. Rather than operating with these rationals directly, the aim is to replace them with good, reduced precision estimates V', V'_ℓ and V'_r, respectively.

Observe that

$$r_o(4V) = \begin{cases} 2\lfloor 2V \rfloor, & \text{if } 2V = \lfloor 2V \rfloor \\ 2\lfloor 2V \rfloor + 1, & \text{otherwise} \end{cases}$$

and analogously for $r_o(4V_\ell)$ and $r_o(4V_r)$.

Now, if $V' \geq V$ is a good overestimate of V, the chance that $\lfloor 2V' \rfloor = \lfloor 2V \rfloor$ is fairly high. The hope is that using V' in place of V produces the same results. Only when V is just below and V' is just above a common integer does equality not hold. Analogously for a good $V'_\ell \geq V_\ell$ and a good $V'_r \geq V_r$.

To pursue this observation further, it is in principle possible for every *fp* to determine how close $2V$, $2V_\ell$ and $2V_r$ come to be integers. With much more sophistication than an unfeasible enumeration over all *fp*s, Dmitry Nadezhin was able to find bounds on these distances and to prove them correct with a certified program written in ACL2, a mechanical theorem prover ([Nadezhin]).

This is an important achievement, as it is the turning point for using estimates of known, reduced precision.

Result 20 (Nadezhin). *Over the range of all* `double`*s with $\epsilon = 2^{-64}$ or over the range of all* `float`*s with $\epsilon = 2^{-32}$*

$$\begin{array}{lcl} \lfloor 2V \rfloor + \epsilon < 2V < (\lfloor 2V \rfloor + 1) - \epsilon & \vee & 2V = \lfloor 2V \rfloor \\ \lfloor 2V_\ell \rfloor + \epsilon < 2V_\ell < (\lfloor 2V_\ell \rfloor + 1) - \epsilon & \vee & 2V_\ell = \lfloor 2V_\ell \rfloor \\ \lfloor 2V_r \rfloor + \epsilon < 2V_r < (\lfloor 2V_r \rfloor + 1) - \epsilon & \vee & 2V_r = \lfloor 2V_r \rfloor \end{array}$$

In other words, there's a 2ϵ-wide off-limit zone around integers where $2V$ can never be found, except when it is itself an integer. (In fact, the off-limit zone is even wider.) This means that if $2V'$ is less than ϵ apart from $2V$, then their floors are the same, regardless of whether $2V$ is an integer or not (recall that $V' \geq V$). Similarly for $2V_\ell$ and $2V_r$.

To show this, assume v to be a `double` or a `float`. Let V' be such that $0 \leq V' - V < \epsilon/2$.

- When $2V - \lfloor 2V \rfloor < 1 - \epsilon$, it follows that $2V' < 2V + \epsilon < \lfloor 2V \rfloor + 1$.
- Otherwise R20 yields $2V = \lfloor 2V \rfloor$ and this leads to $2V' < 2V + \epsilon = \lfloor 2V \rfloor + \epsilon < \lfloor 2V \rfloor + 1$.

In both cases the conclusion is that $\lfloor 2V' \rfloor \leq \lfloor 2V \rfloor$. On the other hand, $V \leq V'$ entails $\lfloor 2V \rfloor \leq \lfloor 2V' \rfloor$ and hence $\lfloor 2V' \rfloor = \lfloor 2V \rfloor$.

The same argument applied to V_ℓ and V_r leads to:

Result 21. *For all* `double`*s or all* `float`*s, with ϵ as in R20*

$$\begin{array}{lcl} 0 \leq V' - V < \epsilon/2 & \Rightarrow & \lfloor 2V' \rfloor = \lfloor 2V \rfloor \\ 0 \leq V'_\ell - V_\ell < \epsilon/2 & \Rightarrow & \lfloor 2V'_\ell \rfloor = \lfloor 2V_\ell \rfloor \\ 0 \leq V'_r - V_r < \epsilon/2 & \Rightarrow & \lfloor 2V'_r \rfloor = \lfloor 2V_r \rfloor \end{array}$$

In other words, “good” overestimates in the sense of this result make the equalities above true. Later in §9.8.3, such good estimates will be defined.

Note that only the right $<$ in the inequalities of R20 are exploited in the discussion above. The left $<$ will be used right below.

9.6 Estimates require a modified rounding

To determine whether the estimates can replace the original full values, suppose $0 \le V' - V < \epsilon/2$, which implies $\lfloor 2V \rfloor = \lfloor 2V' \rfloor$ by R21. Unfortunately, when $2V \in \mathbb{Z}$ it is quite possible that $2V' \notin \mathbb{Z}$, so $r_\mathrm{o}(4V) \neq r_\mathrm{o}(4V')$.

Observe, however, that $2V' < 2V + \epsilon$, so $2V'$ is never too far from $2V$. With this in mind, it is possible to define a slightly modified “almost round-to-odd”:

Definition 9.

$$r'_\mathrm{o}(x) = \begin{cases} 2\lfloor x/2 \rfloor, & \text{if } x/2 - \lfloor x/2 \rfloor < \epsilon \\ 2\lfloor x/2 \rfloor + 1, & \text{otherwise} \end{cases}$$

Stated otherwise, the strict equality in the branching condition of r_o is replaced by another one in r'_o which triggers the upper branch more often.

- Now, suppose at first that $2V \in \mathbb{Z}$. By R21, $\lfloor 2V' \rfloor = \lfloor 2V \rfloor = 2V$ and note that $2V' < 2V + \epsilon$, so $2V' < \lfloor 2V' \rfloor + \epsilon$ and the upper branch of r'_o applies, giving rise to $r'_\mathrm{o}(4V') = 2\lfloor 2V' \rfloor = 2\lfloor 2V \rfloor = r_\mathrm{o}(4V)$.
- Otherwise $2V \notin \mathbb{Z}$. Result 20 leads to $\epsilon < 2V - \lfloor 2V \rfloor$ (the left $<$ in the result is used here). It follows that $2V' - \lfloor 2V' \rfloor = 2V' - \lfloor 2V \rfloor \ge 2V - \lfloor 2V \rfloor > \epsilon$, so $2V' > \lfloor 2V' \rfloor + \epsilon$. The lower branch of r'_o applies: $r'_\mathrm{o}(4V') = 2\lfloor 2V' \rfloor + 1 = 2\lfloor 2V \rfloor + 1 = r_\mathrm{o}(4V)$.

The same also applies to $r_\mathrm{o}(4V_\ell)$ and $r_\mathrm{o}(4V_r)$:

Result 22. *With r'_o as in definition 9:*

$$\bar{v}_\ell = r_\mathrm{o}(4V_\ell) = r'_\mathrm{o}(4V'_\ell) \qquad \bar{v} = r_\mathrm{o}(4V) = r'_\mathrm{o}(4V') \qquad \bar{v}_r = r_\mathrm{o}(4V_r) = r'_\mathrm{o}(4V'_r)$$

For the sake of completeness, it must be added that r'_o *does not*, in general, enjoy the property $x \lesseqgtr 2h \Leftrightarrow r'_\mathrm{o}(x) \lesseqgtr 2h$ which holds for r_o. However, this doesn't matter for the present purposes because r'_o is applied only according to the result above, not to an arbitrary x.

9.7 Another way to express r'_o

While some considerations are simpler when r'_o is expressed as in definition 9, it can be rewritten in another form.

Any x lies between to adjacent even integers, that is, $2h \le x < 2(h+1)$ for some h, which is the same as $h \le x/2 < h+1$. It is clear that $\lfloor x \rfloor = 2h$ when $x < 2h+1$ and $\lfloor x \rfloor = 2h+1$ otherwise, and that $\lfloor x/2 \rfloor = h$ in any case. The branching condition is the same as $2h \le x < 2h + 2\epsilon$.

- When the upper branch is used, as $2\epsilon \le 1$ certainly holds, it follows that $\lfloor x \rfloor = 2h = r'_\mathrm{o}(x)$.
- Otherwise the lower branch is used, so $r'_\mathrm{o}(x) = 2h + 1$.
  - If $x < 2h+1$ then $2h + 1 = \lfloor x \rfloor + 1 = \lfloor x \rfloor \,\square\, 1$, because $i + 1 = i \,\square\, 1$ when i is even.

– When $x \geq 2h+1$ then $2h+1 = (2h+1) \mathbin{[\,]} 1 = \lfloor x \rfloor \mathbin{[\,]} 1$, because $i = i \mathbin{[\,]} 1$ if i is odd.

Result 23.

$$r'_{\mathrm{o}}(x) = \begin{cases} \lfloor x \rfloor, & \textit{if } x/2 - \lfloor x/2 \rfloor < \epsilon \\ \lfloor x \rfloor \mathbin{[\,]} 1, & \textit{otherwise} \end{cases}$$

This form is the one used in the code.

9.8 The estimates V', V'_ℓ and V'_r in practice

The last piece, now, is the determination of the estimates V', V'_ℓ and V'_r. For simplicity, only the case for `double`s and $D = 10$ is assumed from now on.

9.8.1 Rewriting the boundaries of R_v

Given $v = c \otimes 2^q$, the `long` c and the `int` q are almost trivially determined from the bits of v. The definitions of v_ℓ and v_r involve the fractional quantities c_ℓ and c_r of (2) and (3), resp. It's easy to rewrite them, as well as c and q, to involve integers analogues: simply multiply by 4.

$$\bar{c}_\ell = \begin{cases} 4c - 2 & \text{(regular spacing)} \\ 4c - 1 & \text{(irregular spacing)} \end{cases} \qquad \bar{c} = 4c \qquad \bar{c}_r = 4c + 2 \qquad \bar{q} = q - 2$$

Then, of course, $v_\ell = \bar{c}_\ell 2^{\bar{q}}$, $v = \bar{c} 2^{\bar{q}}$, $v_r = \bar{c}_r 2^{\bar{q}}$. Moreover,

$$\bar{c}_\ell < \bar{c} < \bar{c}_r = 4c_r < 4 \cdot 2^P = 2^{P+2} = 2^{55} \tag{8}$$

Each of $\bar{c}_\ell$, $\bar{c}$, $\bar{c}_r$ comfortably fits in a `long`.

9.8.2 Overestimates of powers of 10

Consider $V = v10^{-k} = c2^q10^{-k}$. In general, 10^{-k} requires high or even infinite precision, so the first step is to define an estimate for it of the form $g2^r$. A single Java `long` for g is too narrow to emulate sufficient precision, but two `long`s turn out to be enough to attain R21. By a simple encoding, two `long`s can accommodate non-negative integers of $2 \cdot 63 = 126$ bits.

There are unique β and r such that $10^{-k} = \beta 2^r$ and $2^{125} \leq \beta < 2^{126}$, namely $r = \lfloor \log_2 10^{-k} \rfloor - 125$ and $\beta = 2^{-r}10^{-k}$. Let $g = \lfloor \beta \rfloor + 1$, so $(g-1)2^r \leq 10^{-k} < g2^r$, with the latter value being a pretty good overestimate for 10^{-k}.

The computation of any g is usually expensive but the whole repertoire for `double`s can be precomputed and stored in a lookup table. A quick check in the table further reveals that $g < 2^{126}$ always holds (by the definition alone, the case $g = 2^{126}$ is not excluded), so two `long`s per entry are sufficient.

The whole table thus has $K_{\max} - K_{\min} + 1 = 617$ entries with $2 \cdot 617 = 1\,234$ `long`s, taking up a space of just less than 10 kB. A table of the exponents r is unnecessary, as these are cheap to compute (see the definition of r and §9.1).

9.8.3 V', V'_ℓ and V'_r

Inspired by $V_r = c_r 2^q(\beta 2^r)$, define $V'_r = c_r 2^q(g2^r)$. Note that $V'_r - V_r = c_r 2^{q+r}(g-\beta)$. Because $0 < g - \beta \leq 1$ this gives $0 < V'_r - V_r \leq c_r 2^{q+r}$.

From (4) and (5) we have $2^q 10^{-k} < 2^4$. The definition of r means that $2^{r+125} \leq 10^{-k}$. Multiply by 2^{q-125} to obtain $2^{q+r} < 2^4 \cdot 2^{-125} = 2^{-121}$. Since $c_r < 2^{53}$ it follows that $V'_r - V_r \leq c_r 2^{q+r} < 2^{53} \cdot 2^{-121} = 2^{-68} < \epsilon/2$, as required by R21.

Because $c_\ell < c < c_r < 2^{53}$, the same reasoning also applies to the overestimates V'_ℓ and V'. By simple rewritings we get

Result 24. *The definitions*

$$r = \lfloor \log_2 10^{-k} \rfloor - 125 \qquad g = \lfloor 2^{-r} 10^{-k} \rfloor + 1,$$
$$V'_\ell = (\bar{c}_\ell g) 2^{\bar{q}+r} \qquad V' = (\bar{c} g) 2^{\bar{q}+r} \qquad V'_r = (\bar{c}_r g) 2^{\bar{q}+r}$$

ensure that R21 can be applied. The 126 *bit values for each g are precomputed and stored in a lookup table, two* `long`*s per entry, namely $g_1 = g \mathbin{/\!\!/} 2^{63}$ (the high bits) and $g_0 = g \mathbin{\backslash\!\!\backslash} 2^{63}$ (the lower* 63 *bits).*

9.9 Computation of $\bar{v}$, $\bar{v}_\ell$ and $\bar{v}_r$ in Java

As just mentioned, in a Java implementation the value of each g is held in a pair of non-negative `long`s, each of 63 bits.

Result 22 states that $\bar{v} = r'_o(4V')$. Result 23 needs $\lfloor 4V' \rfloor$ and the fractional part $2V' - \lfloor 2V' \rfloor$ of $2V'$, where $2V' = (\bar{c}g)2^{\bar{q}+r+1} = (\bar{c}g)/2^{-(\bar{q}+r+1)}$. As will be seen shortly, the exponent $-(\bar{q}+r+1)$ is positive, so the divisor is an integer, showing that the fractional part of $2V'$ is represented by the $-(\bar{q}+r+1)$ least significant bits of the integer $\bar{c}g$.

To see that this exponent is definitely positive, it's easy to see from (4) and (5) that $1 \leq 2^q 10^{-k} < 2^4$ always holds, hence $0 \leq q + \lfloor \log_2 10^{-k} \rfloor \leq 3$. This implies $122 \leq -(q+r) \leq 125$, so we get $123 \leq -(\bar{q}+r+1) \leq 126$.

To simplify later computations, however, it's useful to have a fixed width fractional part. Below it will become clear that a good choice for the fractional width is 128 bits. This is easily achieved by left shifting $\bar{c}g$ by the difference $h = 128 - (-(\bar{q}+r+1)) = 128 + \bar{q} + (\lfloor \log_2 10^{-k} \rfloor - 125) + 1$ and hence:

$$h = q + \lfloor \log_2 10^{-k} \rfloor + 2 \tag{9}$$

giving $2 \leq h \leq 5$. As a consequence, with $c' = \bar{c}2^h$, we have $2V' = (c'g)/2^{128}$, so the fractional part of $2V'$ is represented by the 128 least significant bits of $c'g$. Also note that $c' < 2^{60}$ (see (8)), and similarly when $\bar{c}$ gets replaced by $\bar{c}_\ell$ and $\bar{c}_r$, leading to

Result 25. *Let $c'_\ell = \bar{c}_\ell 2^h$, $c' = \bar{c}2^h$ and $c'_r = \bar{c}_r 2^h$. Then*

$$c'_\ell < c' < c'_r < 2^{60} \qquad 2V' = (c'g)2^{-128}$$
$$4V'_\ell = (c'_\ell g)2^{-127} \qquad 4V' = (c'g)2^{-127} \qquad 4V'_r = (c'_r g)2^{-127}$$

With this information at hand, the determination of $\bar{v} = r'_o(4V')$ can proceed by *bit twiddling* as follows:

- Let $b = c'g < 2^{60}\ 2^{126} = 2^{186}$ and define

$$b_{n_0}^{n_1} = (b \mathbin{\backslash\!\!\backslash} 2^{n_1}) \mathbin{/\!\!/} 2^{n_0}$$

that is, the integer consisting of the bits with (0-based) position between n_0 (least significant, included) and n_1 (excluded).
- It's easy to show that

$$\lfloor b2^{-n} \rfloor = b_n^{186} \qquad b2^{-n} - \lfloor b2^{-n} \rfloor = b_0^n 2^{-n}$$
$$b_{n_0}^{n_1} \mathbin{/\!\!/} 2^n = b_{n_0+n}^{n_1} \qquad b_{n_0}^{n_1} \mathbin{\backslash\!\!\backslash} 2^n = b_{n_0}^{n_0+n}\ (n \le n_1 - n_0)$$

- Consequently, $2V' - \lfloor 2V' \rfloor = b2^{-128} - \lfloor b2^{-128} \rfloor = b_0^{128} 2^{-128}$
- The test $2V' - \lfloor 2V' \rfloor < \epsilon$ in r'_{o}, where $\epsilon = 2^{-64}$, becomes

$$b_0^{128} 2^{-128} < 2^{-64} \quad \Leftrightarrow \quad b_0^{128} 2^{-64} < 1 \quad \Leftrightarrow \quad \lfloor b_0^{128} 2^{-64} \rfloor = 0$$

Since $\lfloor b_0^{128} 2^{-64} \rfloor = b_0^{128} \mathbin{/\!\!/} 2^{64} = b_{64}^{128}$, we further get

$$2V' - \lfloor 2V' \rfloor < \epsilon \qquad \Leftrightarrow \qquad b_{64}^{128} = 0$$

The choice of 128 bits for the fractional width thus helps in simplifying this test to the bare minimum once b_{64}^{128} is known.
- Now, $b = c'g = c'(g_1 2^{63} + g_0) = c'g_1 2^{63} + c'g_0$. The products $c'g_1$ and $c'g_0$, do *not* fit in `long`s, so split them into a higher and a lower part of 64 bits:

$$x_1 = c'g_0 \mathbin{/\!\!/} 2^{64} \qquad x_0 = c'g_0 \mathbin{\backslash\!\!\backslash} 2^{64}$$
$$y_1 = c'g_1 \mathbin{/\!\!/} 2^{64} \qquad y_0 = c'g_1 \mathbin{\backslash\!\!\backslash} 2^{64}$$
$$c'g_1 = y_1 2^{64} + y_0 \qquad c'g_0 = x_1 2^{64} + x_0$$

Note that

$$0 \le x_1, y_1 < 2^{59} \qquad 0 \le x_0, y_0 < 2^{64} \qquad 2^h \mid x_0 \qquad 2^h \mid y_0$$

as is easy to prove. They all fit in "unsigned" `long`s (that is, `long`s interpreted as unsigned values). Continuing, we get

$$b = y_1 2^{127} + y_0 2^{63} + x_1 2^{64} + x_0$$

and observing that $y_0/2 \in \mathbb{Z}$, it follows that

$$\begin{aligned} b_{64}^{186} &= b \mathbin{/\!\!/} 2^{64} = \lfloor (y_1 2^{127} + y_0 2^{63} + x_1 2^{64} + x_0) 2^{-64} \rfloor \\ &= y_1 2^{63} + y_0/2 + x_1 + \lfloor x_0 2^{-64} \rfloor \\ &= y_1 2^{63} + y_0/2 + x_1 = y_1 2^{63} + (y_0 \mathbin{/\!\!/} 2 + x_1) \\ &= y_1 2^{63} + z \end{aligned}$$

where

$$z = y_0 \mathbin{/\!\!/} 2 + x_1 = y_0/2 + x_1$$

Note that x_0 disappears: indeed, it is not used at all, not even later, so it's not computed. Also, $z = y_0 \mathbin{/\!\!/} 2 + x_1 < 2^{63} + 2^{59} < 2^{64}$, meaning that it fits in an "unsigned" `long`.

- Further, define

$$\begin{aligned}
\bar{v}' &= \lfloor 4V' \rfloor = \lfloor b2^{-127} \rfloor = b_{127}^{186} = b_{64}^{186} \sslash 2^{63} \\
&= (y_1 2^{63} + z) \sslash 2^{63} \\
&= y_1 + z \sslash 2^{63}
\end{aligned}$$

  Clearly, $\bar{v}' < 2^{59} + 1 < 2^{60}$, so it fits in a `long` as well.
- As shown above, the test in $r_\mathrm{o}'(4V')$ needs b_{64}^{128}:

$$b_{64}^{128} = b_{64}^{186} \bbslash 2^{64} = (y_1 2^{63} + z) \bbslash 2^{64}$$

  Split z into its lower 63 bits and the higher ones:

$$\begin{aligned}
z &= (z \sslash 2^{63}) 2^{63} + z \bbslash 2^{63} \\
&= (\bar{v}' - y_1) 2^{63} + z \bbslash 2^{63}
\end{aligned}$$

  and use this in the equation above:

$$\begin{aligned}
b_{64}^{128} &= (y_1 2^{63} + (\bar{v}' - y_1) 2^{63} + z \bbslash 2^{63}) \bbslash 2^{64} \\
&= (\bar{v}' 2^{63} + z \bbslash 2^{63}) \bbslash 2^{64}
\end{aligned}$$

  Now split $\bar{v}'$ into its lower bit and the higher ones:

$$\bar{v}' = (\bar{v}' \sslash 2) 2 + \bar{v}' \bbslash 2$$

  and use the splitting:

$$\begin{aligned}
b_{64}^{128} &= (((\bar{v}' \sslash 2) 2 + \bar{v}' \bbslash 2) 2^{63} + z \bbslash 2^{63}) \bbslash 2^{64} \\
&= ((\bar{v}' \sslash 2) 2^{64} + (\bar{v}' \bbslash 2) 2^{63} + z \bbslash 2^{63}) \bbslash 2^{64} \\
&= ((\bar{v}' \bbslash 2) 2^{63} + z \bbslash 2^{63}) \bbslash 2^{64}
\end{aligned}$$

  Since $(\bar{v}' \bbslash 2) 2^{63} + z \bbslash 2^{63} < 2^{63} + 2^{63} = 2^{64}$, it turns out that

$$b_{64}^{128} = (\bar{v}' \bbslash 2) 2^{63} + z \bbslash 2^{63}$$

- As a consequence,

$$2V' - \lfloor 2V' \rfloor < \epsilon \quad \Leftrightarrow \quad b_{64}^{128} = 0 \quad \Leftrightarrow \quad \bar{v}' \bbslash 2 = 0 \ \wedge \ z \bbslash 2^{63} = 0$$

  This leads to

$$\bar{v} = r_\mathrm{o}'(4V') = \begin{cases} \bar{v}', & \text{if } \bar{v}' \bbslash 2 = 0 \ \wedge \ z \bbslash 2^{63} = 0 \\ \bar{v}' \talloblong 1, & \text{otherwise} \end{cases}$$

- When $z \bbslash 2^{63} = 0$, observe that $\bar{v}' \bbslash 2 \neq 0$ gives $r_\mathrm{o}'(4V') = \bar{v}' \talloblong 1 = \bar{v}'$ and that $\bar{v}' \bbslash 2 = 0$ yields $r_\mathrm{o}'(4V') = \bar{v}'$ as well.
  Otherwise $z \bbslash 2^{63} \neq 0$ and $r_\mathrm{o}'(4V') = \bar{v}' \talloblong 1$.
  The test can therefore be simplified

$$\bar{v} = r_\mathrm{o}'(4V') = \begin{cases} \bar{v}', & \text{if } z \bbslash 2^{63} = 0 \\ \bar{v}' \talloblong 1, & \text{otherwise} \end{cases}$$

```
MASK_63 = (1L ≪ 63) - 1                          2^63 − 1
x1 = Math.multiplyHigh(g0, cp)                   x_1
y0 = g1 * cp                                      y_0
y1 = Math.multiplyHigh(g1, cp)                   y_1
z = (y0 ⋙ 1) + x1                                z
vbp = y1 + (z ⋙ 63)                              v̄'
vb = vbp | (z & MASK_63) + MASK_63 ⋙ 63          v̄
```

Figure 8: Computing $\bar{v}$ in Java double (similarly for $\bar{v}_\ell$ and $\bar{v}_r$)

- extract c and q from the bits of v
- if $c < C_{\text{tiny}}$ set $c \leftarrow 10c$ and $\Delta k \leftarrow -1$, otherwise set $\Delta k \leftarrow 0$
- set $out \leftarrow c \mathbin{\backslash\!\backslash} 2$ (for *roundTiesToEven*)
- compute k by R14 or R15
- determine g by table lookup and compute h by (9) and R16
- compute $\bar{c}$, $\bar{c}_\ell$ and $\bar{c}_r$ as in §9.8.1
- compute $c' = \bar{c}2^h$, $c'_\ell = \bar{c}_\ell 2^h$, $c'_r = \bar{c}_r 2^h$
- compute $\bar{v} = r'_{\text{o}}(4V')$ as in F8 (§9.9)
- do the same for $\bar{v}_\ell = r'_{\text{o}}(4V'_\ell)$ and for $\bar{v}_r = r'_{\text{o}}(4V'_r)$
- compute $s = \bar{v} \mathbin{/\!\!/} 4$ and $t = s + 1$
- compute $10s' = s - s \mathbin{\backslash\!\backslash} 10$ and $10t' = 10s' + 10$ when needed
- replace $u' \in R_v$ with $\bar{v}_\ell + out \leq 4(10s')$, $w' \in R_v$ with $4(10t') + out \leq \bar{v}_r$
- replace $u \in R_v$ with $\bar{v}_\ell + out \leq 4s$, $w \in R_v$ with $4t + out \leq \bar{v}_r$
- replace $v - u \lesseqgtr w - v$ with $\bar{v} \lesseqgtr 2(s + t)$

Figure 9: The efficient computations for Schubfach

- Finally, noting that

$$(z \mathbin{\backslash\!\backslash} 2^{63} + (2^{63} - 1)) \mathbin{/\!\!/} 2^{63} = \begin{cases} 0, & \text{if } z \mathbin{\backslash\!\backslash} 2^{63} = 0 \\ 1, & \text{otherwise} \end{cases}$$

all this can be rewritten as the conditional-free one-liner

$$\bar{v} = r'_{\text{o}}(4V') = \bar{v}' \mathbin{[\!]} (z \mathbin{\backslash\!\backslash} 2^{63} + (2^{63} - 1)) \mathbin{/\!\!/} 2^{63}$$

Translating these pieces into Java leads to the code in F8. As Java longs are signed, unsigned right-shifts ($\ggg$) are used because the values must be interpreted as “unsigned” longs.

Everything holds similarly for V'_ℓ and V'_r to get $\bar{v}_\ell$ and $\bar{v}_r$, respectively. Indeed, the Java code above is factored out in a method which is invoked three times.

9.10 Roundup

Gathering the pieces together brings to the efficient computations for Schubfach summarized in F9. With these provisions, the Schubfach algorithm of F7 can be translated straightforwardly into Java by any competent programmer.

10 Digits extraction

Digits are extracted without resorting to division. Schubfach returns $d_v = \bar{f}10^{\bar{b}}$, with $\bar{f} < 10^H$. This is first normalized as $d_v = f10^{b-H}$ with $10^{H-1} \le f < 10^H$, so $d_v = \langle 0.f_1 \dots f_H \rangle 10^b$ where the f_i are the decimal digits of f and $f_1 \ge 1$.

Recalling that $H = 17$, f is split in 3 parts

$$h = f \mathbin{/\!\!/} 10^{16} \qquad m = (f \mathbin{\backslash\!\!\backslash} 10^{16}) \mathbin{/\!\!/} 10^8 \qquad l = f \mathbin{\backslash\!\!\backslash} 10^8$$

giving the highest digit h, the middle 8 digits m and the lower 8 digits l.

The reason for the normalization and the splitting is rather technical: it allows a left-to-right extraction discussed in [BZ2013] without resorting to divisions, while keeping all computations in the `long` and `int` ranges. Even the splitting itself is carried out without divisions, as discussed next. The digits of the exponent for scientific notation are extracted without divisions, too.

10.1 D-based right shifts

In this section forget about the conventions in §2.

Let $K > 0$ be in $\mathbb{R}$, let $D \ge 2$, m, $c < K$ all be in $\mathbb{N}$ and let $q = c \mathbin{/\!\!/} D^m$. This is a D-based right shift of c by m positions, where c is limited by K.

The problem addressed here is to compute q without resorting to division. By pre-computing a lookup table for useful combinations of K and m, it is possible to replace the division with a multiplication and a *binary* shift. This normally pays off because a division is usually quite slower.

To see how this can be done, let $\alpha = D^{-m}$ and note that $q = c \mathbin{/\!\!/} D^m = \lfloor c\alpha \rfloor$. Rather than *dividing* by D^m we *multiply* by α. There's a caveat: α is not expressible in binary when D is not a power of 2 (except when $m = 0$). But since we only need the *floor* of $c\alpha$, we approximate α with some other number L representable in binary and hope that $\lfloor c\alpha \rfloor = \lfloor cL \rfloor$ for all $c < K$, even when $\alpha \ne L$. Provided L is a good approximation of α, this should be achievable for values of m and K of practical interest.

In particular, when L has the form $M2^{-F}$, for some $F, M \in \mathbb{N}$, then $\lfloor cL \rfloor$ becomes $\lfloor cM2^{-F} \rfloor = cM \ggg F$: indeed an integer multiplication and a *binary* right shift, as anticipated.

Given $F \in \mathbb{N}$ (to be determined), let $M = \lceil \alpha 2^F \rceil$ and $L = M2^{-F}$, so L is an (over)estimate of α meeting $L - 2^{-F} < \alpha \le L$. From R27 it follows that

$$q \le c\alpha \le q + 1 - \alpha \qquad cL - c2^{-F} \le c\alpha \le cL \tag{10}$$

which shows that $q \le cL$. When L is close enough to α, that is, when F is sufficiently large, it might also meet

$$cL < q + 1 \tag{11}$$

and the conclusion would be $q = c \mathbin{/\!\!/} D^m = \lfloor cL \rfloor$.

Eqs. (10) show that (i) $c\alpha + \alpha \le q + 1$ and (ii) $cL \le c\alpha + c2^{-F}$. By (ii), meeting (a) $c\alpha + c2^{-F} < q + 1$ would satisfy (11) as well; and by (i), meeting (b) $c\alpha + c2^{-F} < c\alpha + \alpha$ would in turn satisfy (a), so let's try to meet (b) or, equivalently, (c) $2^F > cD^m$. As $K > c$, meeting (d) $2^F \ge KD^m$ would also satisfy (c). That is, any F meeting (d) ensures (11).

Since a smaller F means a smaller M, it pays to keep F minimal, leading to

n	m	F	M
17	8	84	193 428 131 138 340 668
17	1	60	115 292 150 460 684 698
9	8	57	1 441 151 881
9	1	34	1 717 986 919
3	2	17	1 311
2	1	10	103

Figure 10: Values for decimal right shifts ($K = D^n$, $D = 10$)

Result 26. *Let $K > 0$ be in $\mathbb{R}$, let $D \geq 1$, m, $c < K$ all be in $\mathbb{N}$. Define*

$$F = \lceil \log_2(KD^m) \rceil = \lceil \log_2 K + m \log_2 D \rceil \qquad M = \lceil 2^F / D^m \rceil$$

(In particular, when $K = 2^p$ then $F = p + \lceil m \log_2 D \rceil$ and when $K = D^n$ then $F = \lceil (n+m) \log_2 D \rceil$.)

Then

$$c \mathbin{/\!\!/} D^m = \lfloor cM2^{-F} \rfloor = \texttt{c * M >>> F}$$

A table for values used in the Java code appears in F10.

11 Other results

Some auxiliary results are collected here.

Consider $m \in \mathbb{Z}$, $n \in \mathbb{N}^*$ and let $q = m \mathbin{/\!\!/} n$. Clearly, $q \leq m/n < q+1$, so $qn \leq m < (q+1)n$. Since all quantities here are integers, an equivalent formulation is $qn \leq m \leq (q+1)n - 1$, leading to

Result 27. *$m \in \mathbb{Z}$, $n \in \mathbb{N}^*$ and $q = m \mathbin{/\!\!/} n$ entail $q \leq m/n \leq q + 1 - 1/n$.*

Consider $\alpha \in [0, 1)$ and $n \in \mathbb{N}^*$, so clearly $i = \lfloor i + \alpha \rfloor$. Let $e = (i+\alpha) \mathbin{/\!\!/} n$ and $\beta = (i+\alpha) \mathbin{\backslash\!\!\backslash} n$, meaning that $i+\alpha = en+\beta$. Therefore, $i \mathbin{/\!\!/} n = \lfloor i + \alpha \rfloor \mathbin{/\!\!/} n = \lfloor en + \beta \rfloor \mathbin{/\!\!/} n = e + \lfloor \beta \rfloor \mathbin{/\!\!/} n = e$, as n and e are integers and $0 \leq \beta < n$. Also, since $0 \leq \gamma - \lfloor \gamma \rfloor < 1$ always holds, we further get $\lfloor \gamma \rfloor \mathbin{/\!\!/} n = \gamma \mathbin{/\!\!/} n$.

Result 28. *Given $\alpha \in [0, 1)$ and $n \in \mathbb{N}^*$ it follows that $(i + \alpha) \mathbin{/\!\!/} n = i \mathbin{/\!\!/} n$. Consequently, $\gamma \mathbin{/\!\!/} n = \lfloor \gamma \rfloor \mathbin{/\!\!/} n$.*

12 Certified proofs

The results discussed above are meant for *human consumption*. They are formulated and proved on paper; as such, they are subject to unintentional errors and discrepancies. While a lot of care and time was invested to make them sound, they should be taken with a grain of salt.

Mechanical formulations of some proofs exist, however, in form of ACL2 programs. They are meant for *machine consumption*. The degree of confidence afforded by a theorem prover is far greater than with traditional mathematical tools like the paper-and-pencil style used in this writing. On the flip side, a mechanical proof requires even seemingly trivial results to be described with utmost precision, often obscuring the overall picture with a sea of details.

A big thanks therefore goes to the late Dmitry Nadezhin[1] whose untiring willingness to prepare ACL2 proofs for some of the results of this writing ([Nadezhin]) goes far beyond the patience of most of us. Result 20, in addition, *requires* machine assistance to find an appropriate ϵ.

Given that the nature of human-focused and machine-ready proofs is quite different, they were developed almost independently. This might add even more confidence in both the proofs and their results.

13 Timeline

The quest for accurate rendering of floating-point values started with the seminal paper of Steele and White ([SW1990]), followed by several iterations of more efficient algorithms. For further notes about earlier related work, see [Ada2018].

An earlier design similar to, but predating Grisu ([Loi2010]), implemented around 2004 by this author to overcome well-known bugs of the JDK implementation, was kept unpublished for many years for lack of time to productize it. It was eventually polished and submitted to the OpenJDK `core-libs-dev` mailing list much later, in April 2018 ([Giu2018A]). Similarly to Grisu, it makes use of reduced precision arithmetic in about 99.4% of the cases and resorts to full precision for the remaining ones. It guarantees uncompromising results as by specification.

Shortly thereafter, reviewing the accompanying paper, Dmitry Nadezhin observed that with sufficient, yet limited precision, a novel design could get rid of full precision arithmetic altogether. Schubfach was born and implemented as a non-iterative design. Dmitry's insights cannot be emphasized enough.

A first variant of the code presented in this writing has been submitted to the `core-libs-dev` mailing list in September 2018 but did not attract much interest, partly because the subject is often wrongly perceived as obscure and for specialists only, but mainly because of lack of this accompanying documentation ([Giu2018B]). Admittedly, it is hard to make much sense of the code's core without further explanations.

The submission, however, was noticed by Adams, who published a paper about his own Ryū design ([Ada2018]) just a few months earlier. This author was made aware of Ryū by its designer on that same mailing list. Surprisingly and interestingly, Ryū is based on the same observation that is at the core of Schubfach efficient rewritings: a precomputed table of powers of 10, each of sufficient yet rather limited precision, opens the way to efficient arithmetic *over the whole range of* `double`*s*. Ryū, however, makes use of an iterative search.

14 Appendix: Schubfach for `float`s

The table of powers of 10 from R24 remains the same but is accessed differently because a lesser precision is sufficient.

More precisely, there are unique real β' and integer r' such that $10^{-k} = \beta' 2^{r'}$ with $2^{62} \le \beta' < 2^{63}$. It follows that $r' = \lfloor \log_2 10^{-k} \rfloor - 62$ and $\beta' = 2^{-r'} 10^{-k}$.

[1] Hi Dima, please allow me to dedicate this writing to your memory.

```
MASK_32 = (1L << 32) - 1                               2^32 − 1
x1 = Math.multiplyHigh(g, cp)                          x_1
vbp = x1 >>> 31                                        v̄'
vb = vbp | (x1 & MASK_32) + MASK_32 >>> 32             v̄
```

Figure 11: Computing $\bar{v}$ in Java float (similarly for $\bar{v}_\ell$ and $\bar{v}_r$)

Also, let $g' = \lfloor\beta'\rfloor + 1$. Given g_1 and g_0 as defined in R24, it's not hard to show that $g' = g_1 + 1$, since $g_0 > 0$ for all entries.

The details are not worked out here, but a line of reasoning analogous to, but somewhat simpler than the one used in §9.9, leads to a width of the fractional part of $2V'$ of 96 bits, which gives rise to

$$h = q + \lfloor\log_2 10^{-k}\rfloor + 33$$

and to the even simpler code in F11.

The nice thing about floats is that *all* 2^{32} results of toString can be checked extensively to fully meet the specification in less than a couple of hours. This has been performed successfully several times during development.

References

[Ada2018] Adams, "Ryū: Fast Float-to-String Conversion", *PLDI*, 2018, 270–282

[BM2005] Boldo & Melquiond, "When double rounding is odd", *17th IMACS World Congress*, 2005, HAL Id: inria-00070603

[BZ2013] Bouvier & Zimmermann, *Division-Free Binary-to-Decimal Conversion*, INRIA, 2013, HAL Id: hal-00864293

[Giu2018A] Giulietti, `https://mail.openjdk.java.net/pipermail/core-libs-dev/2018-April/052696.html`

[Giu2018B] Giulietti, `https://mail.openjdk.java.net/pipermail/core-libs-dev/2018-September/055698.html`

[Loi2010] Loitsch, "Printing floating-point numbers quickly and accurately with integers", *PLDI*, 2010, 233–243

[Mat1968] Matula, "In-and-out Conversions", *CACM*, 1968, 11(1), 47–50

[MBD+2018] Muller et al, *Handbook of Floating-point Arithmetic (2nd ed)*, Birkhäuser, 2018

[Nadezhin] Nadezhin, `https://github.com/nadezhin/verify-todec`

[SW1990] Steele & White, "How to Print Floating-Point Numbers Accurately", *ACM SIGPLAN Notices*, 1990, 112–123