BASIC COMPUTER ARITHMETIC

TSOGTGEREL GANTUMUR

Abstract. First, we consider how integers and fractional numbers are represented and manipulated internally on a computer. The focus is on the principles behind the algorithms, rather than on implementation details. Then we develop a basic theoretical framework for analyzing algorithms involving inexact arithmetic.

Contents

1. Introduction

There is no way to encode all real numbers by using finite length words, even if we use an alphabet with countably many characters, because the set of finite sequences of integers is countable. Fortunately, the real numbers support many countable dense subsets, and hence the encoding problem of real numbers may be replaced by the question of choosing a suitable countable dense subset. Let us look at some practical examples of real number encodings.

- Decimal notation. Examples: 36000, 2.35(75), −0.000072.
- Scientific notation. Examples: $3.6\cdot10^4$, $-72\cdot10^{-6}$. The general form is $m\cdot10^e$. In order to have a unique (or near unique) representation of each number, one can impose a *normalization*, such as requiring $1 \le |m| < 10$.
- System with base/radix β. Example: $m_2m_1m_0.m_{-1} = m_2\beta^2 + m_1\beta + m_0 + m_{-1}\beta^{-1}$. The dot separating the integer and fractional parts is called the *radix point*.
- Binary ($\beta = 2$), octal ($\beta = 8$), and hexadecimal ($\beta = 16$) numbers.
- Babylonian hexagesimal ($\beta = 60$) numbers. This format is "true floating point," in the sense that no radix point is used, but the position of the radix point is inferred from context. For example, $46:2:16 = (46\cdot60^2 + 2\cdot60 + 16)\cdot60^k$ where k is arbitrary.
- Mixed radix. Example: $13_{(360)}10_{(20)}3 = 13\cdot360 + 10\cdot20 + 3$ (Mayan calendar format).
- Rationals: $\frac{p}{q}$, $p \in \mathbb{Z}$, $q \in \mathbb{N}$, $q \ne 0$. This builds on a way to represent integers.
- Continued fraction: $[4;2,1,3] = 4 + \frac{1}{2+\frac{1}{1+\frac{1}{3}}}$.
- Logarithmic number systems: b^x, where $x \in \mathbb{R}$ is represented in some way.
- More general functions: $\sqrt{2}$, e^2, $\sin 2$, $\arctan 1$.

Date: May 7, 2018.

As history has shown, simple base-β representations (i.e., place-value systems) seem to be best suited for general purpose computations, and they are accordingly used in practically all modern digital computing devices. Inevitably, not only real numbers must be approximated by fractional numbers that are admissible on the particular device, but also the operations on the admissible numbers themselves should be approximate. Although the details of how fractional numbers are implemented may vary from device to device, it is possible to formulate a general set of assumptions encompassing practically all implementations, which are still specific enough so as not to miss any crucial details of any particular implementation.

In what follows, we first describe how integers and fractional numbers are handled on majority of modern computing devices, and will then be naturally led to the aforementioned set of assumptions, formulated here in the form of two axioms. Once the axioms have been formulated, all the precursory discussions become just one (albeit the most important) example satisfying the axioms. Finally, we illustrate by examples how the axioms can be used to analyze algorithms that deal with real numbers, which gives us an opportunity to introduce several fundamental concepts of numerical analysis.

2. Integers

Given a *base* (or *radix*) $\beta \in \mathbb{N}$, $\beta \geq 2$, any integer $a \in \mathbb{Z}$ can be expressed as

$$a = \pm \sum_{k=0}^{\infty} a_k \beta^k, \tag{1}$$

where $0 \leq a_k \leq \beta - 1$ is called the *k-th digit of a in base β*. The digits can also be thought of as elements of $\mathbb{Z}/\beta\mathbb{Z}$, the integers modulo β. Obviously, each integer has only finitely many nonzero digits, and the digits are *uniquely* determined by a. More precisely, we have

$$a_k = \left\lfloor \frac{|a|}{\beta^k} \right\rfloor \mod \beta, \tag{2}$$

where $\lfloor x \rfloor = \max\{n \in \mathbb{Z} : n \leq x\}$ is the *floor function*.

In modern computers, we have $\beta = 2$, i.e., *binary numbers* are used, mainly because it simplifies hardware implementation. Recall that one binary digit is called a *bit*. At the hardware level, modern CPU's handle 64 bit sized integers, which, in the nonnegative (or *unsigned*) case, would range from 0 to $M-1$, where $M = 2^{64} \approx 18 \cdot 10^{18}$. Note that in embedded systems, 16 bit (or even 8 bit) microprocessors are more common, so that $M = 2^{16} = 65536$. There are several methods to encode *signed* integers, with the most popular one being the so called *two's complement*. This makes use of the fact that $M - a \equiv -a \mod M$ and hence $M - a$ behaves exactly like $-a$ in the modulo M arithmetic:

$$\begin{aligned} (M-a) + b &= M + b - a \equiv b - a \mod M, \\ (M-a) \cdot b &= Mb - ab \equiv -ab \mod M, \\ (M-a) \cdot (M-b) &= M^2 - M(a+b) + ab \equiv ab \mod M, \\ (M-a)b + r = c \quad &\iff \quad -ab + r \equiv c \mod M, \quad \text{etc.} \end{aligned} \tag{3}$$

Thus for the 64 bit architecture, the signed integers would be $\tilde{\mathbb{Z}} = \{-2^{63}, \dots, 2^{63}-1\}$, where the numbers $2^{63}, \dots, 2^{64}-1$ are *internally* used to represent $-2^{63}, \dots, -1$. Note that $2^{63} \approx 9 \cdot 10^{18}$. Within either of the aforementioned signed and unsigned ranges, each arithmetic operation takes a few clock cycles, with the multiplication and division being the most expensive. Hence it makes sense to measure the complexity of an algorithm by how many multiplication and division operations it needs to execute (unless of course the algorithm has disproportionately many addition and/or subtraction, in which case the relevant operations would be those). If

the result of an operation goes out of the admissible range, a flag (or exception) called *integer overflow* would be raised. The other type of error one should watch out for is *division by zero*.

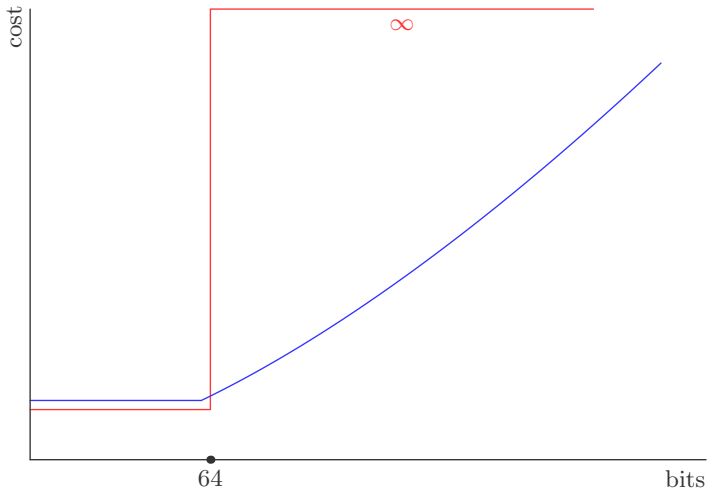


FIGURE 1. The fixed length of standard integers can be modelled by assigning infinite cost to any operation associated to integers longer than the standard size (red curve). For platforms that can handle arbitrarily large integers (bignums), the cost of each arithmetic operation grows depending on the length of the input variables (blue curve).

Integers outside of the usual ranges are called *bignums*, and can be handled in software by representing them as arrays of "digits," where a single "digit" could now hold numbers up to $\sim 2^{64}$ or $\sim 10^6$ etc., since the digit-by-digit operations would be performed by the built-in arithmetic. In other words, we use (1) with a large β. There is no such thing as "bignum overflow," in the sense that the allowed size of a bignum would only be limited by the computer memory. Programming languages such as Python, Haskell, and Ruby have built-in implementations of bignum arithmetic. Obviously, the complexity of an algorithm involving bignum variables will depend on the lengths of those bignum variables. For example, addition or subtraction of two bignums of respective lengths n and m has the complexity of $O(n+m)$ in general. This type of complexity measure is usually called *bit complexity*.

$$\begin{array}{r} {\scriptstyle 1\ 1\ \ 1\ \ } \\ 7725 \\ +\ 5308 \\ \hline 13033 \end{array} \qquad \begin{array}{r} {\scriptstyle \ \ 1\ 1\ \ } \\ 1286 \\ -\ 473 \\ \hline 813 \end{array} \qquad \begin{array}{r} 4653 \\ \times\ \ 207 \\ \hline 32571 \\ 9306\ \cdot \\ \hline 963171 \end{array} \qquad \begin{array}{r|l} 438 & 17 \\ \cline{2-2} -\ 34\ \ & 25 \\ \cline{1-1} \ \ 98 & \\ -\ 85 & \\ \cline{1-1} \ \ 13 & \end{array}$$

FIGURE 2. Illustration of the grade school algorithms.

When two integers are given by their digits, elementary arithmetic operations can be performed by employing the usual algorithms that we learn in grade school. This is relevant to both built-in and bignum arithmetics. First, we can reduce the general case into a case where the two integers are *nonnegative*. Then the digits of the sum $s = a + b$ are given by the recurrent relation

$$c_k\beta + s_k = a_k + b_k + c_{k-1}, \qquad k = 0, 1, \ldots, \tag{4}$$

where $0 \le s_k \le \beta - 1$, and $c_{-1} = 0$. Since $a_0 + b_0 \le \beta + (\beta - 2)$, we have $c_0 \le 1$, and by induction, all the "carry digits" satisfy $c_k \le 1$. The digit-wise operations such as (4) are performed either by elementary logic circuits in case our goal is the built-in arithmetic, or by the built-in arithmetic in case we are talking about implementing a bignum arithmetic.

To compute the difference $d = a - b$, we can always assume $a \ge b$, and the digits of d are

$$d_k = c_k\beta + a_k - b_k - c_{k-1}, \qquad k = 0, 1, \dots, \tag{5}$$

where the "borrowed digits" c_k are uniquely determined by the condition $0 \le d_k \le \beta - 1$, and $c_{-1} = 0$. It is straightforward to show that $0 \le c_k \le 1$.

Addition and subtraction can also be treated simultaneously as follows. First, we introduce the intermediate representation, which we call the *Cauchy sum or difference*:

$$a \pm b = \sum_{k=0}^{\infty} x_k^* \beta^k, \tag{6}$$

with $x_k^* = a_k \pm b_k \in \{1 - \beta, \dots, 2\beta - 2\}$. This is not a standard representation because we may have $x_k^* < 0$ or $x_k^* > \beta - 1$. The true digits of $x = a \pm b$ can be found from

$$c_k\beta + x_k = x_k^* + c_{k-1} \qquad k = 0, 1, \dots, \tag{7}$$

where the integers $c_k \in \{-1, 0, 1\}$ are uniquely determined by the condition $0 \le x_k \le \beta - 1$, and $c_{-1} = 0$. As long as we ensure that both numbers are nonnegative, and that $a \ge b$ in case of subtraction, the result will have an expansion with finitely many nonzero digits.

To multiply two positive integers, we first define the *Cauchy product*

$$ab = \Big(\sum_{j=0}^{\infty} a_j\beta^j\Big) \cdot \Big(\sum_{i=0}^{\infty} b_i\beta^i\Big) = \sum_{k=0}^{\infty} \Big(\sum_{j=0}^{k} a_j b_{k-j}\Big)\beta^k = \sum_{k=0}^{\infty} p_k^*\beta^k, \tag{8}$$

where

$$p_k^* = \sum_{j=0}^{k} a_j b_{k-j}, \tag{9}$$

is the k-th *generalized digit* of ab. In general, p_k^* can be larger than $\beta - 1$, and so (8) is not the base-β expansion of the product ab. However, the proper digits $0 \le p_k \le \beta - 1$ of ab can be found by

$$c_k\beta + p_k = p_k^* + c_{k-1} \qquad k = 0, 1, \dots, \tag{10}$$

where $c_{-1} = 0$. One way to find the base-β expansion of p_k^* would be to do the summation in (9) from the beginning in base-β arithmetic.

Preceding algorithm generates the digits of the product ab directly, i.e., it implements the grade school algorithm *column-wise*. We can also approach it *row-wise*, as

$$ab = \Big(\sum_{j=0}^{n} a_j\beta^j\Big) \cdot b = \sum_{j=0}^{n} a_j \cdot \beta^j b, \tag{11}$$

where $\beta^j b$ can be produced by simple shifts, and multiplication by the single digit a_j can be implemented in terms of additions and shifts. The radix-2 case ($\beta = 2$) is especially simple: Set $\pi_0 = 0$, and for $j = 0, \dots, n$, perform

$$\pi_{j+1} = \begin{cases} \pi_j & \text{if } a_j = 0, \\ \pi_j + 2^j b & \text{if } a_j = 1. \end{cases} \tag{12}$$

Then the final result π_{n+1} is equal to the product ab. So an adder (in combination with bit shifts) is all one needs in order to implement this algorithm.

We can write the same process, starting with the most significant bit, instead of the least significant one, as follows. Let $\pi_0 = 0$, and for $j = 0, \dots, n$, let

$$\pi_{j+1} = \begin{cases} \pi_j & \text{if } a_{n-j} = 0, \\ \pi_j + 2^{n-j} b & \text{if } a_{n-j} = 1. \end{cases} \tag{13}$$

The intermediate results π_j are called *partial products* or *interim products*. The recurrence (13) has a nice graphical representation, see Figure 3 (left). We will not discuss accelerated hardware implementations of the aforementioned addition and multiplication algorithms, as it would open a whole other can of worms.

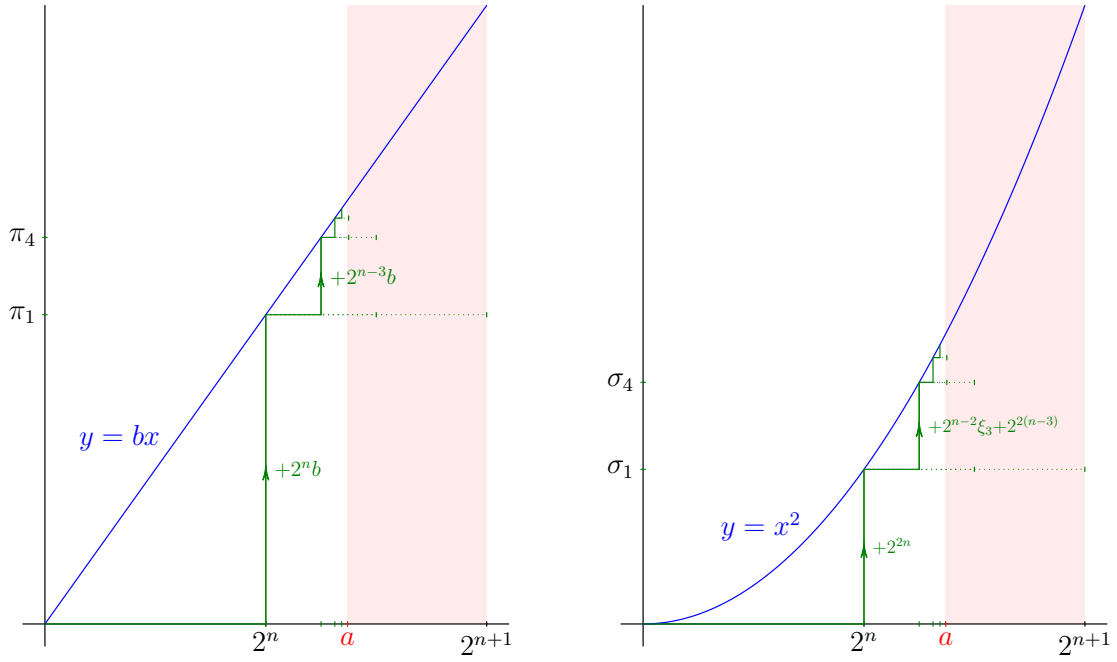


FIGURE 3. Binary multiplication and bit-wise squaring.

Before closing this section, we present an algorithm to compute the square a^2, that reads the bits of a one by one. Let ξ_j denote the number formed by nullifying $n+1-j$ least significant bits of a. For example, if $a = 1101$ and so $n = 3$, then $\xi_0 = 0$, $\xi_1 = 1000$, $\xi_2 = \xi_3 = 1100$, and $\xi_4 = 1101$. Supposing that the partial square ξ_j^2 has been computed, and that $a_j = 1$, we have the update formula

$$\xi_{j+1}^2 = (\xi_j + 2^{n-j})^2 = \xi_j^2 + 2 \cdot 2^{n-j} \cdot \xi_j + 2^{2(n-j)}, \tag{14}$$

which leads to the following algorithm. Set $\sigma_0 = \xi_0 = 0$, and for $j = 0, \dots, n$, perform

$$(\sigma_{j+1}, \xi_{j+1}) = \begin{cases} (\sigma_j, \xi_j) & \text{if } a_{n-j} = 0, \\ (\sigma_j + 2^{n+1-j}\xi_j + 2^{2(n-j)}, \xi_j + 2^{n-j}) & \text{if } a_{n-j} = 1. \end{cases} \tag{15}$$

The process is illustrated in Figure 3 (right). Notice the similarities between (13) and (15). Although one would probably never implement (15) in practice, it hints at a general class of algorithms for evaluating functions, based on addition laws, such as

$$f(x + 2^{-j}) = f(x) + g(x, 2^{-j}), \tag{16}$$

where $g(x, 2^{-j})$ is easily computable, either directly or from tabulated values. This class of algorithms usually goes under the name *pseudo-multiplications*.

As discussed, the bit complexity of addition and subtraction is $O(n+m+1)$, where

$$n = \max\{k : a_k \neq 0\}, \qquad m = \max\{k : b_k \neq 0\}. \tag{17}$$

Note that "+1" is introduced into $O(\dots)$ because according to the way n and m are defined, the *lengths* of a and b are $n+1$ and $m+1$, respectively. On the other hand, it is easy to see that the same for our multiplication algorithm is $O(nm+1)$. In fact, there exist asymptotically much faster multiplication algorithms such as the Karatsuba algorithm and the Schönhage-Strassen algorithm, with the latter having the bit complexity of $O(n+m+1)$ up to some logarithmic factors.

3. Simple division algorithms

Now we turn to division. Generally, the exact quotient of two integers is a fractional number. One can stop the division process at any stage, and return the approximate quotient and the remainder. It will be convenient to assume that the quotient is expanded as

$$q = q_0 + q_{-1}\beta^{-1} + q_{-2}\beta^{-2} + \dots, \tag{18}$$

where $0 \leq q_j < \beta$ are the digits. Some division algorithms use generalized digits that satisfy a more relaxed condition such as $-\beta < q_j < \beta$, cf. (6) and (8). In order to have the quotient in the above form, we assume that a and b are positive and normalized, such that

$$a = a_0 + a_{-1}\beta^{-1} + a_{-2}\beta^{-2} + \dots, \qquad b = b_0 + b_{-1}\beta^{-1} + b_{-2}\beta^{-2} + \dots, \tag{19}$$

with $a_0 > 0$ and $b_0 > 0$, that is, we assume that $1 \leq a < \beta$ and $1 \leq b < \beta$. This normalization can be achieved by pre-scaling, and one needs to do the corresponding adjustments to the final quotient after the division has been done.

Restoring division. The division algorithms we discuss here are only effective for small values of the radix β, and for simplicity, we consider the case $\beta = 2$ first. The first algorithm is called the *restoring division*, and works as follows. We compute $a - b$, and if $a - b \geq 0$, then we set $q_0 = 1$ and $a' = a - b$. In this case, since $a < 2$ and $b \geq 1$, we have $a' = a - b < 1 \leq b$. On the other hand, if $a - b < 0$, then we set $q_0 = 0$ and $a' = a$. In this case, we also have $a' = a < b$. Now we apply the same process to a' and $b' = \frac{1}{2}b$, to compute the digit q_{-1}. This process is repeated until the partial remainder becomes small enough. More formally, the algorithm sets $\alpha_0 = a$, and for $j = 0, 1, \dots$, performs

$$(\alpha_{j+1}, q_{-j}) = \begin{cases} (\alpha_j - 2^{-j}b, 1) & \text{if } \alpha_j - 2^{-j}b \geq 0, \\ (\alpha_j, 0) & \text{if } \alpha_j - 2^{-j}b < 0. \end{cases} \tag{20}$$

The quantities α_j are called *partial remainders*. The process is illustrated in Figure 4(a).

We have $0 \leq \alpha_0 = a < 2 \leq 2b$. Taking $0 \leq \alpha_j < 2^{1-j}b$ as the induction hypothesis, we have

$$0 \leq \alpha_{j+1} = \alpha_j - 2^{-j}b < 2^{1-j}b - 2^{-j}b = 2^{-j}b, \tag{21}$$

for the first branch of (20), while for the second branch $0 \leq \alpha_{j+1} = \alpha_j < 2^{-j}b$ holds trivially. By construction, we have

$$\alpha_j = q_{-j}2^{-j}b + \alpha_{j+1}, \tag{22}$$

and so

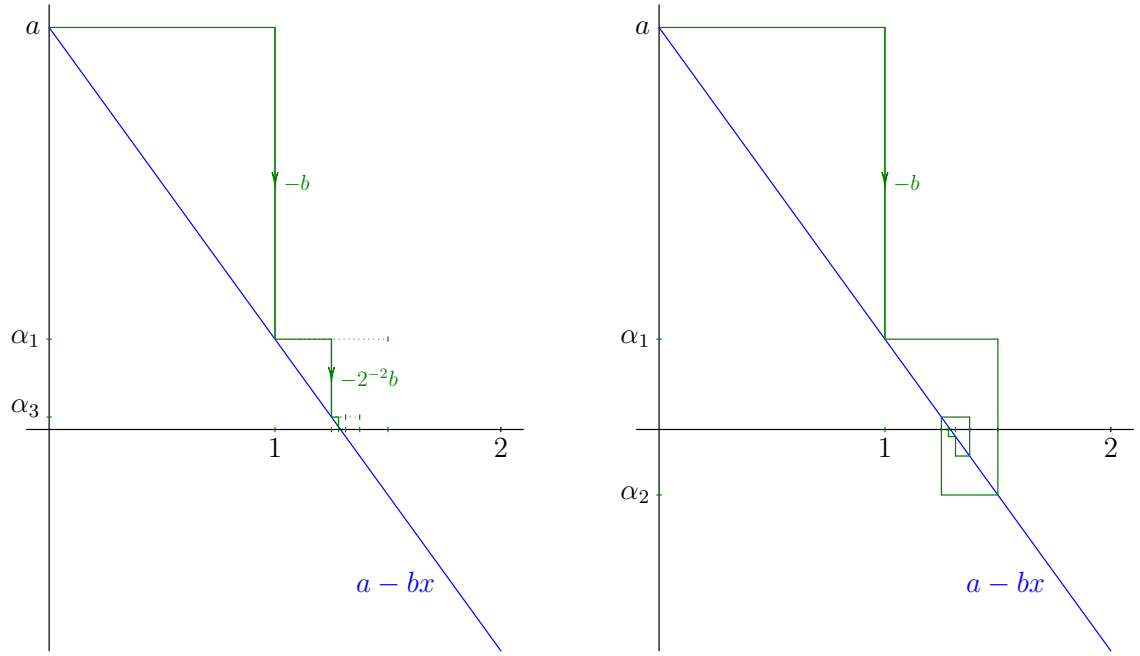
$$\begin{aligned} a = \alpha_0 &= q_0 b + \alpha_1 = q_0 b + q_{-1}2^{-1}b + \alpha_2 = \dots \\ &= q_0 b + q_{-1}2^{-1}b + \dots + q_{-n}2^{-n}b + \alpha_{n+1} \\ &= qb + \alpha_{n+1}, \end{aligned} \tag{23}$$

where $q = q_0 + q_{-1}2^{-1} + \dots + q_{-n}2^{-n}$. In light of the bound $0 \leq \alpha_{n+1} = \alpha_j < 2^{-n}b$, this implies

$$0 \leq \frac{a}{b} - q < 2^{-n}, \tag{24}$$

meaning that the digits of q are precisely the first $n+1$ digits of the quotient a/b.

Since the algorithm generates one correct bit per step, it terminates after n steps, where n is how many quotient digits we need. Furthermore, each step costs us a bit shift, a subtraction, and a comparison operation.



(A) $q = 101001\ldots$ (radix point omitted). (B) $q = 11\bar{1}1\bar{1}\bar{1}\ldots$, where $\bar{1} = -1$.

FIGURE 4. Restoring and non-restoring divisions.

In practical implementations of (20), it is more convenient to scale up the partial remainders, than to scale down the divisor. Thus, putting $\alpha_j = 2^{-j} r_j$ into (20), we get the recurrence

$$(r_{j+1}, q_{-j}) = \begin{cases} (2(r_j - b), 1) & \text{if } r_j - b \geq 0, \\ (2r_j, 0) & \text{if } r_j - b < 0. \end{cases} \tag{25}$$

for $j = 0, 1, \ldots$. On the other hand, the scaling $\alpha_j = 2^{1-j} r_j$ would give

$$(r_{j+1}, q_{-j}) = \begin{cases} (2r_j - b, 1) & \text{if } 2r_j - b \geq 0, \\ (2r_j, 0) & \text{if } 2r_j - b < 0, \end{cases} \tag{26}$$

for $j = 0, 1, \ldots$, which is more standard in the literature. Note that in the latter case, the initialization becomes $r_0 = 2^{-1} a$ (or equivalently, b is replaced by $2b$). If one computes the difference $2r_j - b$ and overwrites its value into $2r_j$, then in case $2r_j - b < 0$, one has to "restore" the value $2r_j$ by adding b back. This is the reason for the name "restoring division."

Non-restoring division. The point of the restoring step is of course to prevent the partial remainders from becoming negative. If we skip a restoring step, therefore allowing negative partial remainders, then the subsequent iterations of the division process must try to drive the partial remainders to 0, by *adding* multiples of b. This idea is graphically illustrated in Figure 4(b). A way to record the fact that we added a multiple of b to the current partial remainder, as opposed to subtracting, is to set the corresponding quotient digit to -1. In other words, instead of the standard bits $q_j \in \{0, 1\}$, we now have the generalized digits $q_j \in \{-1, 1\}$,

and a conversion to the standard binary representation is needed as a post-processing step. This leads to the following recurrence. Let $\alpha_0 = a$, and let

$$(\alpha_{j+1}, q_{-j}) = \begin{cases} (\alpha_j - 2^{-j}b, 1) & \text{if } \alpha_j \geq 0, \\ (\alpha_j + 2^{-j}b, -1) & \text{if } \alpha_j < 0, \end{cases} \tag{27}$$

for $j = 0, 1, \ldots$. The analysis follows exactly the same lines as that of the restoring division. First, since $1 \leq a, b < 2$, we have $-2b < \alpha_0 = a < 2 \leq 2b$. Now, taking $-2^{1-j}b \leq \alpha_j < 2^{1-j}b$ as the induction hypothesis, we infer

$$\begin{aligned} -2^{-j}b \leq \alpha_{j+1} = \alpha_j - 2^{-j}b < 2^{1-j}b - 2^{-j}b = 2^{-j}b \qquad &\text{if } \alpha_j \geq 0, \\ 2^{-j}b > \alpha_{j+1} = \alpha_j + 2^{-j}b \geq -2^{1-j}b + 2^{-j}b = -2^{-j}b \qquad &\text{if } \alpha_j < 0, \end{aligned} \tag{28}$$

which shows that $-2^{1-j}b \leq \alpha_j < 2^{1-j}b$ for all j. By construction, we have

$$\alpha_j = q_{-j}2^{-j}b + \alpha_{j+1}, \tag{29}$$

and so

$$a = \alpha_0 = q_0 b + q_{-1}2^{-1}b + \ldots + q_{-n}2^{-n}b + \alpha_{n+1} = qb + \alpha_{n+1}, \tag{30}$$

where $q = q_0 + q_{-1}2^{-1} + \ldots + q_{-n}2^{-n}$. Then from $\alpha_{n+1} = \frac{a}{b} - q$, we deduce

$$0 \leq \tfrac{a}{b} - q < 2^{-n}, \qquad \text{if} \qquad \alpha_{n+1} \geq 0, \tag{31}$$

$$2^{-n} \leq \tfrac{a}{b} - q < 0, \qquad \text{if} \qquad \alpha_{n+1} < 0. \tag{32}$$

Therefore, if $\alpha_{n+1} \geq 0$, then the digits of q are the first $n+1$ digits of the quotient a/b. If $\alpha_{n+1} < 0$ then the simple modification $q' = q - 2^{-n}$ satisfies

$$0 \leq \tfrac{a}{b} - q' < 2^{-n}, \tag{33}$$

and hence q' yields the first $n+1$ digits of a/b.

For a practical implementation, the scaling $\alpha_j = 2^{1-j}r_j$ gives

$$(r_{j+1}, q_{-j}) = \begin{cases} (2r_j - b, 1) & \text{if } r_j \geq 0, \\ (2r_j + b, -1) & \text{if } r_j < 0, \end{cases} \tag{34}$$

for $j = 0, 1, \ldots$, with the initialization $r_0 = 2^{-1}a$. To compute the standard binary representation of q, we can simply separate the generalized digits q_j according to $q_j = 1$ or $q_j = -1$, and subtract. More precisely, let $q_j^+ = \max\{q_j, 0\}$, and $q_j^- = \max\{-q_j, 0\}$, and let

$$\begin{aligned} q^+ &= q_0^+ + q_{-1}^+ 2^{-1} + \ldots + q_{-n}^+ 2^{-n}, \\ q^- &= q_0^- + q_{-1}^- 2^{-1} + \ldots + q_{-n}^- 2^{-n}. \end{aligned} \tag{35}$$

Then $q = q^+ - q^-$. For example, omitting the radix point, if the generalized digits of q generated by (34) are $q = 11\bar{1}1\bar{1}\bar{1}$, where $\bar{1} = -1$, then $q^+ = 110100$ and $q^- = 001011$, and so $q = q^+ - q^- = 101001$, cf. Figure 4. Since either $q_j = 1$ or $q_j = -1$, we always have $q_j^+ + q_j^- = 1$, which implies that $q^+ + q^- = 2^{n+1} - 1$, after a scaling. Thus we have

$$q = q^+ - q^- = q^+ - (2^{n+1} - 1 - q^+) = 2q^+ + 1 - 2^{n+1} = 2q^+ + 1 \qquad (\mathrm{mod}\ 2^{n+1}), \tag{36}$$

that is, q is simply a bit shift (to the left) of q^+, with the spill-over bit beyond the position n discarded, and the least significant bit set to 1. For the aforementioned example $q^+ = 110100$, writing 1 on the right of it, we have 1101001, and removing the leftmost bit, we get $q = 101001$. This suggests a strategy to find the standard bits of q directly within the algorithm, by looking ahead at the sign of r_{j+1}, instead of the current partial remainder r_j. Moreover, supposing that q has been converted into its standard binary representation, note that the modification

$q' = q - 2^{-n}$ in (33) is simply a matter of setting $q_{-n} = 0$. Therefore, this step can be interpreted as shifting the (implicit) bit q^+_{-n-1} to the left. All in all, we conclude that the recurrence

$$r_{j+1} = \begin{cases} 2r_j - b & \text{if } r_j \geq 0, \\ 2r_j + b & \text{if } r_j < 0, \end{cases} \qquad q_{-j} = \begin{cases} 1 & \text{if } r_{j+1} \geq 0, \\ 0 & \text{if } r_{j+1} < 0, \end{cases} \tag{37}$$

for $j = 0, 1, \ldots$ computes the digits of the quotient $2r_0/b$, provided $1 \leq 2r_0, b < 2$.

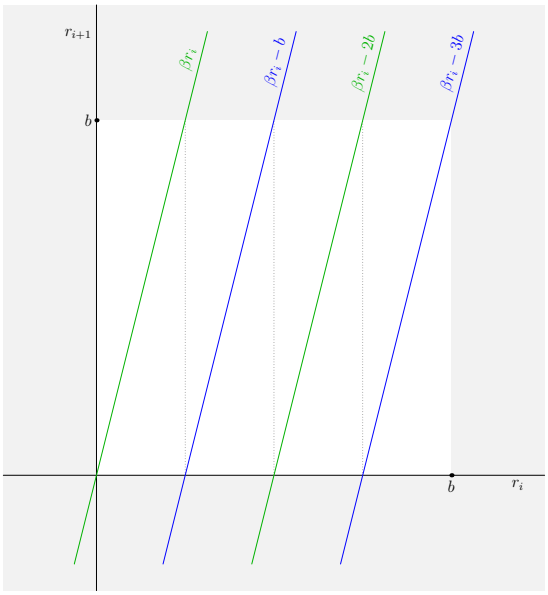
General radix. In principle, it is not hard to generalize the aforementioned division algorithms to any radix β. For restoring division, the partial remainders are given by

$$r_{j+1} = \beta r_j - q_{-j} b, \qquad j = 0, 1, \ldots, \tag{38}$$

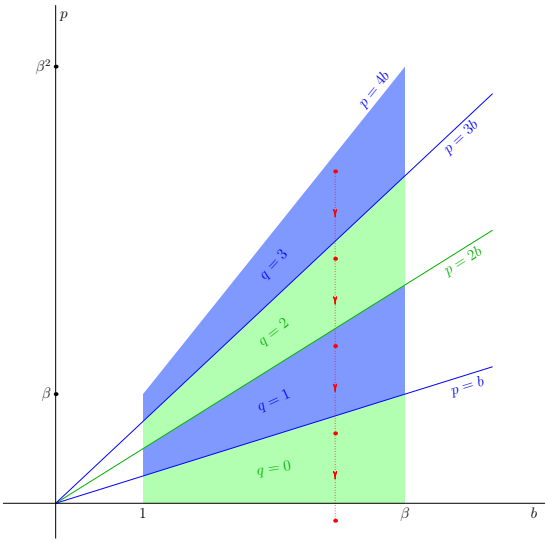
where the quotient digits satisfy

$$q_{-j} = \left\lfloor \frac{\beta r_j}{b} \right\rfloor, \tag{39}$$

see Figure 5. A straightforward approach to determine q_{-j} is to subtract multiples of b from βr_j, until it is in the range $[0, b)$. For example, in the case $\beta = 4$, it involves computing $\beta r_j - b$, $\beta r_j - 2b$, and $\beta r_j - 3b$. In general, this approach can be arranged to that the quotient digit is determined in approximately $\log_2 \beta$ additions/subtractions and comparison operations.



(A) Given the partial remainder r_i, the quotient digit q_{-i} is chosen according to which of the slanted lines falls inside the white square. This ensures that the condition $0 \leq r_i < b$ is preserved.



(B) Dependence of the quotient digit $q = q_{-i}$ on the partial remainder $p = \beta r_i$ and the divisor b. All possible combinations (b, p) are represented by the coloured trapezoidal area. The red dots illustrate repeated subtractions of b in a particular situation.

FIGURE 5. Radix-4 restoring division.

Returning to the analysis of (38), we have

$$\begin{aligned} r_0 &= \beta^{-1} q_0 b + \beta^{-1} r_1 = \beta^{-1} q_0 b + \beta^{-2} q_{-1} b + \beta^{-2} r_2 = \ldots \\ &= \beta^{-1} q_0 b + \beta^{-2} q_{-1} b + \ldots + \beta^{-n} q_{-n} b + \beta^{-n-1} r_{n+1} \\ &= \beta^{-1} q b + \beta^{-n-1} r_{n+1}, \end{aligned} \tag{40}$$

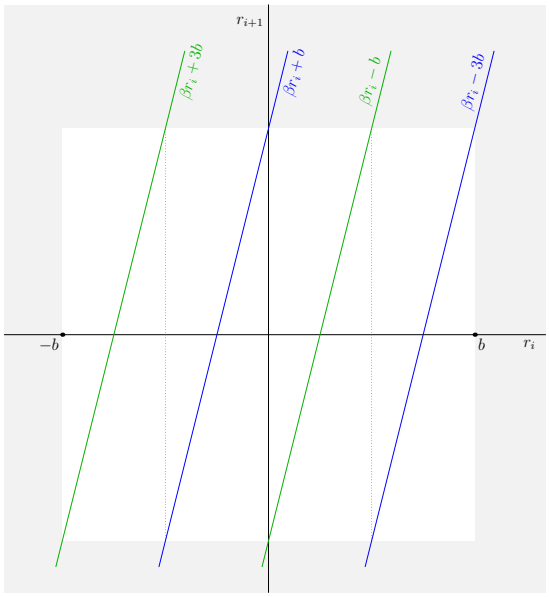
so that

$$\beta r_0 = qb + \beta^{-n} r_{n+1}, \qquad \text{with} \qquad q = q_0 + q_{-1}\beta^{-1} + \ldots + q_{-n}\beta^{-n}. \tag{41}$$

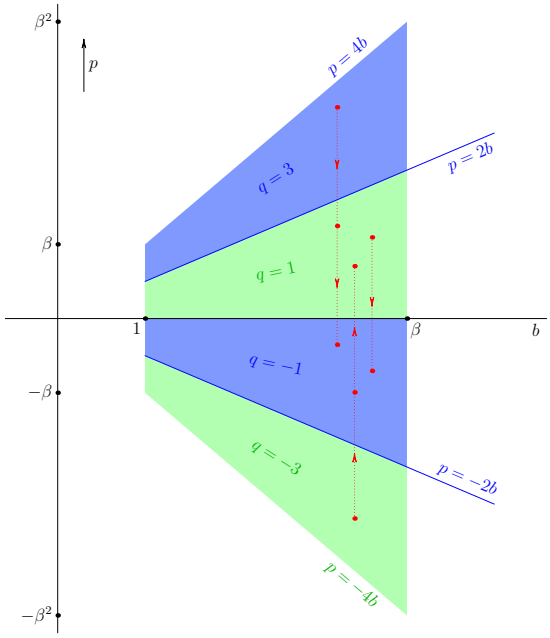
The assumptions $1 \le \beta r_0 < \beta$ and $1 \le b < \beta$ guarantee that $0 \le r_j < b$ and $0 \le q_{-j} \le \beta - 1$ for all j, implying that

$$0 \le \frac{\beta r_0}{b} - q < \beta^{-n}, \tag{42}$$

i.e., the digits of q are in fact the first $n+1$ digits of the quotient $\beta r_0/b$.



(A) Given the partial remainder r_i, the quotient digit q_{-i} is chosen according to which of the slanted lines falls inside the white square. This ensures that the condition $-b \le r_i < b$ is preserved.



(B) Dependence of the quotient digit $q = q_{-i}$ on the partial remainder $p = \beta r_i$ and the divisor b. All possible combinations (b, p) are represented by the coloured trapezoidal area. The red dots illustrate additions/subtractions of $2b$.

FIGURE 6. Radix-4 non-restoring division.

For non-restoring division, the recursion (38) still holds, but the quotient digit computation is different from (39). Namely, negative digits are allowed, and the condition we want to preserve is $-b \le r_j < b$, as opposed to $0 \le r_j < b$. Let us discuss the details for the case $\beta = 4$. One possibility is to allow the digits $\{\pm 1, \pm 3\}$, which will be sufficient to ensure $-b \le r_{j+1} < b$ provided that $-b \le r_j < b$, cf. Figure 6. From the diagram, it is easy to see that

$$q_{-j} = 2\left\lfloor \frac{\beta(r_j + b)}{2b} \right\rfloor - (\beta - 1). \tag{43}$$

Of course, one would not use this formula in implementations; a simple method to find q_{-j} would involve computing the quantities $\beta r_j \pm b$, $\beta r_j \pm 3b$ and checking if they are in the range $[-b, b)$. In the context of (43), if $-b \le r_j < b$, then $0 \le \beta(r_j + b) < 2\beta b$, and so $-(\beta - 1) \le q_{-j} \le \beta - 1$. Moreover, substituting (43) into (38), we get

$$r_{j+1} = \beta(r_j + b) - 2b\left\lfloor \frac{\beta(r_j + b)}{2b} \right\rfloor - b, \tag{44}$$

which makes it clear that $-b \le r_{j+1} < b$. By scaling, we can always arrange $1 \le \beta r_0 < \beta$ and $1 \le b < \beta$, that is, $\beta^{-1} \le r_0 < 1 \le b < \beta$, so that $-b \le r_j < b$ and $-(\beta-1) \le q_{-j} \le \beta - 1$ for all j. Finally, the analysis (40) works here verbatim, which yields

$$\beta r_0 = qb + \beta^{-n} r_{n+1}, \qquad \text{with} \qquad q = q_0 + q_{-1}\beta^{-1} + \ldots + q_{-n}\beta^{-n}, \tag{45}$$

implying that

$$0 \le \tfrac{\beta r_0}{b} - q < \beta^{-n}, \qquad \text{if} \qquad r_{n+1} \ge 0, \tag{46}$$

$$\beta^{-n} \le \tfrac{\beta r_0}{b} - q < 0, \qquad \text{if} \qquad r_{n+1} < 0. \tag{47}$$

Analogously to the binary case, if $r_{n+1} < 0$ then the simple modification $q' = q - \beta^{-n}$ would give us the correct digits of the quotient.

As we have seen, the straightforward way of computing the quotient digits, cf. (39) and (43), presents a bottleneck, even when the radix β is not so large. In the next 2 sections, we will discuss the following approaches to deal with this problem.

- Use only a few digits of βr_j and b to estimate q_{-j}, and correct it (Long division).
- The same as above, but use the estimated digit without correcting (SRT division).

4. Long division

The restoring and non-restoring division algorithms work fine when the radix β, as well as the lengths of a and b are moderate. However, they are not efficient especially for bignums, where both the radix and the length of integers can be large. Even for built-in arithmetic, it would be preferable to have some sort of parallelism between the computation of the next partial remainder and the determination of the next quotient digit. In this section, we are going to build an algorithm based on the usual long division algorithm we study in grade school. What we have in mind here is mainly bignum division, but some of the discussions will be relevant to the next section, where we study a more sophisticated division method for built-in arithmetic.

Our goal is to compute the digits of $q \ge 0$ and $0 \le r < b$, satisfying

$$a = qb + r, \tag{48}$$

that is, we focus on *Euclidean division* of two integers. Here and in what follows, unless otherwise specified, all variables are *integer* variables. Without loss of generality, we can assume that $n \ge m$ and $a > b \ge 2$, where n and m are as defined in (17). By replacing b by $\beta^{n-m}b$, we could even assume $n = m$, but as we will see below, the cases $m = 0$ and $m = 1$ are somewhat special, so we will treat n and m independently.

As a warm up, let us treat the special case $m = 0$ first. In this case, b has only one digit, i.e., $2 \le b \le \beta - 1$, so division can be performed in a straightforward digit-by-digit fashion. This case is sometimes called "short division." The first step of the division algorithm would be to divide a_n by b, as

$$a_n = q_n b + r_n, \tag{49}$$

where $0 \le r_n < b$ is the remainder, and $q_n \ge 0$ is the quotient. Obviously, $q_n \le \beta - 1$ because $a_n \le \beta - 1$. Computation of q_n and r_n should be performed in computer's built-in arithmetic. To proceed further, we combine r_n with the $(n-1)$-st digit of a, and divide it by b, that is,

$$r_n\beta + a_{n-1} = q_{n-1}b + r_{n-1}, \tag{50}$$

where $0 \le r_{n-1} < b$. Since $r_n < b$, we are guaranteed that $q_{n-1} \le \beta - 1$. For bignum arithmetic, computation of q_{n-1} and r_{n-1} should be performed in computer's built-in arithmetic, and since this involves division of the 2-digit number $r_n\beta + a_{n-1}$ by the 1-digit number b, 2-digit

numbers must be within the reach of the built-in arithmetic. More precisely, we need $\beta^2 \le M$. This procedure is repeated until we retrieve the last digit a_0, and we finally get

$$\begin{aligned} a &= a_n\beta^n + \ldots + a_0 = (q_n b + r_n)\beta^n + a_{n-1}\beta^{n-1} + \ldots + a_0 \\ &= q_n b\beta^n + (r_n\beta + a_{n-1})\beta^{n-1} + \ldots + a_0 \\ &= q_n b\beta^n + (q_{n-1}b + r_{n-1})\beta^{n-1} + \ldots + a_0 = \ldots \\ &= q_n b\beta^n + q_{n-1}b\beta^{n-1} + \ldots + q_0 b + r_0 \\ &= b\sum_{k=0}^{n} q_k\beta^k + r_0, \end{aligned} \tag{51}$$

which shows that q_k is the k-th digit of q, and that $r = r_0$.

$$\begin{array}{rl|l} & 9\,2\,5 & 7 \\ \hline - & 7 & 1\,3\,2 \\ \hline & 2\,2 & \\ - & 2\,1 & \\ \hline & 1\,5 & \\ - & 1\,4 & \\ \hline & 1 & \end{array} \qquad \begin{array}{rl|l} & 9\,2\,5 & 9 \\ \hline - & 9 & 1\,0\,2 \\ \hline & 0\,2 & \\ - & 0 & \\ \hline & 2\,5 & \\ - & 1\,8 & \\ \hline & 7 & \end{array} \qquad \begin{array}{rl|l} & 9\,2\,5 & 5 \\ \hline - & 5 & 1\,8\,5 \\ \hline & 4\,2 & \\ - & 4\,0 & \\ \hline & 2\,5 & \\ - & 2\,5 & \\ \hline & 0 & \end{array}$$

FIGURE 7. "Short division": To divide any number by a one-digit number, it is enough to be able to divide any two-digit number by the one-digit number.

With reference to the quotient digit formula (39), we see that the current approach uses only one digit of a at a time, that is, we approximate the partial remainder of the restoring division by its first digit, which still gives the correct quotient digit. This is more efficient than computing with the exact partial remainder. In fact, for the general case $m > 0$, the overall structure of the algorithm is exactly the same. However, since the divisor b can now have a large number of digits, we will also be forced to approximate it by its first few digits. This will make the quotient digit computation inexact, which we will need to deal with in some way. In Figure 8(a), we illustrate the situation where the divisor b has 2 digits, and the partial remainder $p = \beta r_i$ is approximated by its leading 3 digits. In Figure 8(b), the divisor b may have any number of digits, but is approximated by its leading 2 digits, while the partial remainder is approximated by its leading 3 digits. Compare these diagrams with Figure 5(b).

Let us introduce a convenient new notation. For $0 \le k \le \ell$ let

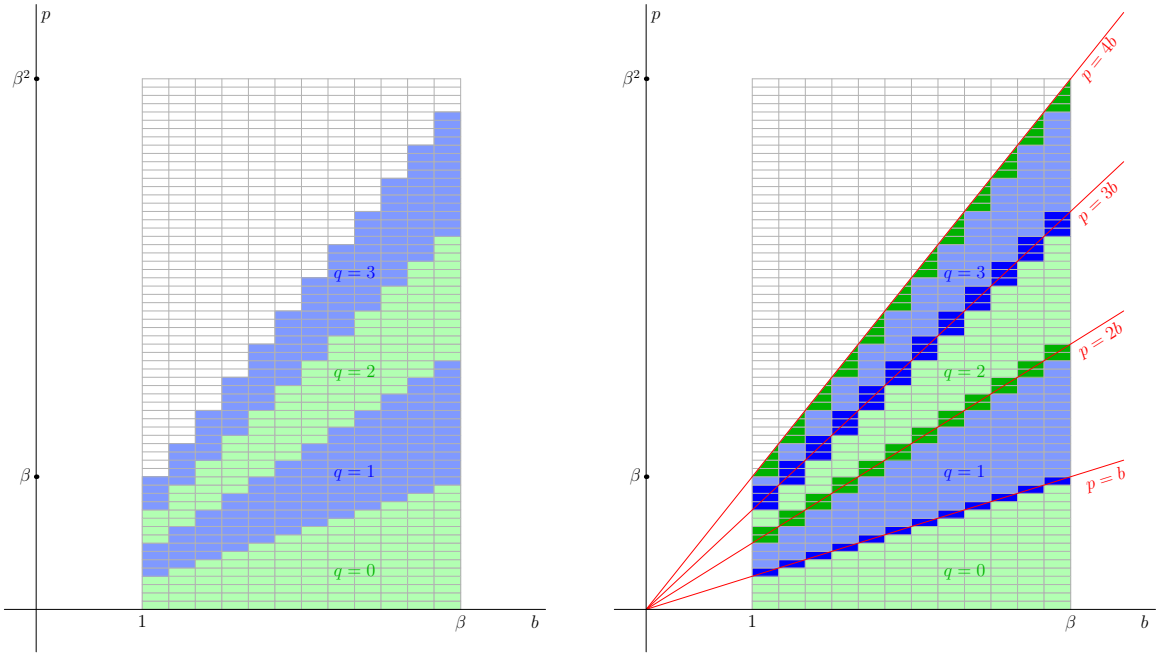
$$a_{[k,\ell]} = a_k + a_{k+1}\beta + \ldots + a_\ell\beta^{\ell-k}, \tag{52}$$

which is simply the number consisting of those digits of a that are numbered by $k, \ldots, \ell$. For example, when $\beta = 10$ and $a = 1532$, we have $a_{[2,4]} = 15$. The first step of our algorithm is to compute q_{n-m} and $0 \le r_{n-m} < b$ satisfying

$$a_{[n-m,n]} = q_{n-m}b + r_{n-m}. \tag{53}$$

Since the number of digits of $a_{[n-m,n]}$ is the same as that of b, we have $q_{n-m} \le \beta - 1$. Next, we compute q_{n-m-1} and $0 \le r_{n-m-1} < b$ satisfying

$$r_{n-m}\beta + a_{n-m-1} = q_{n-m-1}b + r_{n-m-1}. \tag{54}$$



(A) The divisor b is assumed to be a 2-digit number, normalized so that $1 \le b < \beta$. Thus b can take only finitely many values, and so it is clear that the quotient q depends only on the leading 3 digits of p. Note that we have $0 \le p \equiv \beta r_i < \beta b$, cf. Figure 5(b).

(B) Approximate b by its leading 2 digits, and p by 3 digits, i.e., q is computed only at the lower left corner of each small rectangle. Then there is no underestimate, and it is possible to overestimate by 1 only in the rectangles that intersect with the red lines.

FIGURE 8. Radix-4 long division.

Since $r_{n-m} < b$, we are guaranteed that $q_{n-m-1} \le \beta - 1$. We repeat this process until we retrieve the last digit a_0, and as before, we have

$$\begin{aligned} a &= a_{[n-m,n]}\beta^{n-m} + a_{n-m-1}\beta^{n-m-1} + \ldots + a_0 \\ &= (q_{n-m}b + r_{n-m})\beta^{n-m} + a_{n-m-1}\beta^{n-m-1} + \ldots + a_0 \\ &= q_{n-m}b\beta^{n-m} + (r_{n-m}\beta + a_{n-m-1})\beta^{n-m-1} + \ldots + a_0 \\ &= q_{n-m}b\beta^{n-m} + (q_{n-m-1}b + r_{n-m-1})\beta^{n-m-1} + \ldots + a_0 = \ldots \\ &= q_{n-m}b\beta^{n-m} + q_{n-m-1}b\beta^{n-m-1} + \ldots + q_0 b + r_0 \\ &= b\sum_{k=0}^{n-m} q_k\beta^k + r_0, \end{aligned} \tag{55}$$

which shows that q_k is the k-th digit of q, and that $r = r_0$.

This seems all well and good, except that as mentioned before, there is a catch: In (53) and (54), we divide by b, which has $m+1$ digits, and we cannot rely on the built-in arithmetic since m can be large. We encounter the divisions (53) and (54) in each step of the paper-and-pencil long division method. There, what helps is intuition and the fact that in practice we usually have m not too large. Here, we need to replace intuition by a well defined algorithm. We shall consider here an approach that is based on a few crucial observations. The first observation is that since $r_{n-m} < b$ and $a_{n-m-1} < \beta$, we have

$$r_{n-m}\beta + a_{n-m-1} \le (b-1)\beta + \beta - 1 = b\beta - 1, \tag{56}$$

```
  9 2 5 | 7 7
- 7 7   | 1 2
-------
  1 5 5
- 1 5 4
-------
      1
```

```
  8 5 3 5 5 | 1 2 3 6
- 7 4 1 6   | 6 9
-----------
  1 1 1 9 5
- 1 1 1 2 4
-----------
        7 1
```

```
  1 2 3 0 0 4 | 7 7 7 7
- 7 7 7 7     | 1 5
-------------
  4 5 2 3 4
- 3 8 8 8 5
-----------
    6 3 4 9
```

FIGURE 9. Examples of division by multiple-digit numbers. Note that in each stage of each case, the current digit of the result can be accurately estimated by dividing the number formed by the first 2 or 3 digits of the partial remainder, by the number formed by the first 2 digits of the divisor. For instance, in the 1st step of the 2nd case, we have 85/12=7, while the correct digit is 6. In the 2nd stage, we have 111/12=9, which is the correct digit. See also Figure 8(b).

so that the left hand side of (54) has at most $m+2$ digits. Noting that the left hand side of (53) has $m+1$ digits, we now see that (53) and (54) only require divisions of a number not exceeding $b\beta-1$ by b. In other words, *the original division problem a/b has been reduced to the case $a \le b\beta-1$* (and hence with $m \le n \le m+1$). This indeed helps, because if two numbers have roughly the same number of digits, then the first few digits of both numbers can be used to compute a very good approximation of the quotient. For instance, as we shall prove below in Theorem 1, it turns out that under the assumption $a \le b\beta-1$, if

$$a_{[m-1,n]} = q^* b_{[m-1,m]} + r^*, \tag{57}$$

with $0 \le r^* < b_{[m-1,m]}$, then

$$q \le q^* \le q+1, \tag{58}$$

cf. Figure 8(b). This means that the quotient of the number formed by the first 2 or 3 digits of a, divided by the number formed by the first 2 digits of b, is either equal to the quotient q of a divided by b, or off by 1. The cases $q^* = q+1$ can easily be detected (and immediately corrected) by comparing the product q^*b with a. For bignum arithmetic, the division (57) can be performed in the built-in arithmetic, because the number of digits of any of the operands therein does not exceed 3. As this requires that 3-digit numbers (i.e., numbers of the form $c_0 + c_1\beta + c_2\beta^2$) should be within the reach of the built-in arithmetic, we get the bit more stringent condition $\beta^3 \le M$, compared to the previous $\beta^2 \le M$. It is our final requirement on β, and hence, for instance, we can take $\beta = 2^{21}$ or $\beta = 10^6$.

We shall now prove the claim (58).

Theorem 1. *Assume that $2 \le b < a \le b\beta-1$ and hence $m \le n \le m+1$, where n and m are as in (17). Let q and $0 \le r < b$ be defined by*

$$a = qb + r, \tag{59}$$

and with some (small) integer $p \ge 0$, let q^* and $0 \le r^* < b_{[m-p,m]}$ be defined by

$$a_{[m-p,n]} = q^* b_{[m-p,m]} + r^*. \tag{60}$$

Then we have

$$q \le q^* < q + 1 + \beta^{1-p}, \tag{61}$$

and so $q \le q^* \le q+1$ holds as long as $p \ge 1$.

Proof. Let $a^* = a_{[m-p,n]}\beta^{m-p}$, let $b^* = b_{[m-p,n]}\beta^{m-p}$, and redefine r^* to be $r^*\beta^{m-p}$. Then rescaling of the equation (60) by the factor β^{m-p} gives

$$a^* = q^* b^* + r^*, \qquad \text{with} \quad 0 \le r^* \le b^* - \beta^{m-p}. \tag{62}$$

We also have

$$\beta^m \le b^* \le b < b^* + \beta^{m-p} \qquad \text{and} \qquad \beta^n \le a^* \le a < a^* + \beta^{m-p}. \tag{63}$$

We start by writing

$$q - q^* = \frac{a-r}{b} - \frac{a^*-r^*}{b^*} = \frac{a}{b} - \frac{a^*}{b^*} + \frac{r^*}{b^*} - \frac{r}{b}. \tag{64}$$

Then invoking $b \ge b^*$ and $r \ge 0$, we get

$$q - q^* \le \frac{a-a^*}{b^*} + \frac{r^*}{b^*} < \frac{\beta^{m-p} + r^*}{b^*} \le 1, \tag{65}$$

where we have used $a - a^* < \beta^{m-p}$ in the second step, and $r^* \le b^* - \beta^{m-p}$ in the third step. This shows that $q - q^* < 1$, and hence $q \le q^*$.

To get an upper bound on q^*, we proceed as

$$q^* - q = \frac{a^*}{b^*} - \frac{a}{b} + \frac{r}{b} - \frac{r^*}{b^*} \le \frac{a}{b^*} - \frac{a}{b} + \frac{r}{b} = \frac{a}{b} \cdot \frac{b-b^*}{b^*} + \frac{r}{b}, \tag{66}$$

where we have simply used $a^* \le a$ and $r^* \ge 0$. Now, the estimates $a < b\beta$, $b - b^* < \beta^{m-p}$, $b^* \ge \beta^m$, and $r \le b - 1$ give

$$q^* - q < \frac{b\beta}{b} \cdot \frac{\beta^{m-p}}{\beta^m} + \frac{b-1}{b} = \beta^{1-p} + 1 - \frac{1}{b} < \beta^{1-p} + 1, \tag{67}$$

which yields the desired estimate $q^* < q + 1 + \beta^{1-p}$. The proof is complete. □

Finally, let us put the algorithm together in a coherent form.

Algorithm 1: Pseudocode for long division

Data: Integers $a = \sum_{k=0}^{n} a_k \beta^k$ and $b = \sum_{k=0}^{m} b_k \beta^k$ in radix β, satisfying $a > b \ge 2$.
Result: Digits of the quotient q and the remainder r satisfying $a = qb + r$ and $0 \le r < \beta$.
Without loss of generality, assume $a_n \ne 0$ and $b_m \ne 0$.
Compute q^* and $0 \le r^* < b_{[m-1,m]}$ satisfying $a_{[n-1,n]} = q^* b_{[m-1,m]} + r^*$, cf. (53).
if $q^* b \le a_{[n-m,n]}$ **then**
  $q_{n-m} := q^*$
  $r_{n-m} := a_{[n-m,n]} - q_{n-m} b$
else
  $q_{n-m} := q^* + 1$
  $r_{n-m} := a_{[n-m,n]} - q_{n-m} b$
end
for $k := n - m$ **to** 1 **do**
  Let $c := r_k \beta + a_{k-1}$
  if $c = 0$ **then** Set $q_{k-1} := 0$ and $r_{k-1} := 0$, and go to the next iteration.
  Let the expansion of c be $c = \sum_{i=0}^{\ell} c_i \beta^i$ with $c_\ell \ne 0$.
  Compute q^* and $0 \le r^* < b_{[m-1,m]}$ satisfying $c_{[m-1,\ell]} = q^* b_{[m-1,m]} + r^*$, cf. (54).
  if $q^* b \le c$ **then**
    $q_{k-1} := q^*$
    $r_{k-1} := c - q_{k-1} b$
  else
    $q_{k-1} := q^* + 1$
    $r_{k-1} := c - q_{k-1} b$
  end
end

Exercise 1. Estimate the bit complexity of the long division algorithm.

5. The SRT division algorithm

In this section, we consider the so-called SRT division algorithm, named after Sweeney, Robertson, and Tocher. We use the setting from Section 3. The division algorithms we have considered so far can be brought under one roof by writing them as the recurrence

$$r_{j+1} = \beta r_j - q_j b, \tag{68}$$

where q_j are the (generalized) digits of the quotient, as a function of r_j and b, computed in some way, cf. (34), (39), (43), (54), etc. Note that for convenience, we replaced the notation q_{-j} by q_j. Repeating the previous analysis for the general recurrence (68), we infer

$$\begin{aligned} r_0 &= \beta^{-1} q_0 b + \beta^{-1} r_1 = \beta^{-1} q_0 b + \beta^{-2} q_1 b + \beta^{-2} r_2 = \dots \\ &= \beta^{-1} q_0 b + \beta^{-2} q_1 b + \dots + \beta^{-n} q r_n b + \beta^{-n-1} r_{n+1} \\ &= \beta^{-1} q b + \beta^{-n-1} r_{n+1}, \end{aligned} \tag{69}$$

leading to

$$\beta r_0 = q b + \beta^{-n} r_{n+1}, \qquad \text{with} \qquad q = q_0 + q_1 \beta^{-1} + \dots + q_n \beta^{-n}. \tag{70}$$

Therefore, as long as the partial remainders r_n are bounded in absolute value independently of n, the quantity q will approximate the quotient $\beta r_0/b$ better and better as n grows. More precisely, assuming that

$$-R_1 b \le r_n < R_2 b \qquad \text{for all } n, \tag{71}$$

for some constants $R_1, R_2 \ge 0$, we get

$$-\beta^{-n} R_1 \le \frac{\beta r_0}{b} - q < \beta^{-n} R_2, \tag{72}$$

so that each step of the recurrence (68) reduces the error in the quotient β-fold. For restoring division and long division, we have $R_1 = 0$ and $R_2 = 1$, while for non-restoring division, we have $R_1 = R_2 = 1$.

We employed approximations of the partial remainder and the divisor in long division, which resulted in occasional errors that needed to be corrected. The main idea behind the SRT division is to use redundant digits to represent the quotient, so that the correction step is not necessary. Fix a constant $\alpha \in \mathbb{N}$, and suppose that we allow the quotient digits to be $q_i \in \{-\alpha, \dots, \alpha\}$. Then let us try to maintain

$$-Rb \le r_j < Rb \qquad \text{for all } j, \tag{73}$$

for some constant $R > 0$. In other words, we require that for all $-Rb \le r < Rb$, the inequality

$$-Rb \le \beta r - q b < Rb, \tag{74}$$

have at least one solution $q \in \{-\alpha, \dots, \alpha\}$. First of all, a necessary condition is

$$2Rb \ge b \qquad \text{or} \qquad R \ge \frac{1}{2}, \tag{75}$$

since otherwise, choosing r such that $\beta r = Rb$ would give $\beta r - b = (R-1)b < -\frac{1}{2} b < -Rb$, and hence the inequality (74) would have no integer solution. On the other hand, letting r and q take their extreme values, the best bounds on $\beta r - qb$ we can get are

$$-\beta R b + \alpha b \le \beta r - q b < \beta R b - \alpha b, \tag{76}$$

implying that we need

$$\beta R - \alpha \le R \qquad \text{or} \qquad R \le \frac{\alpha}{\beta - 1}. \tag{77}$$

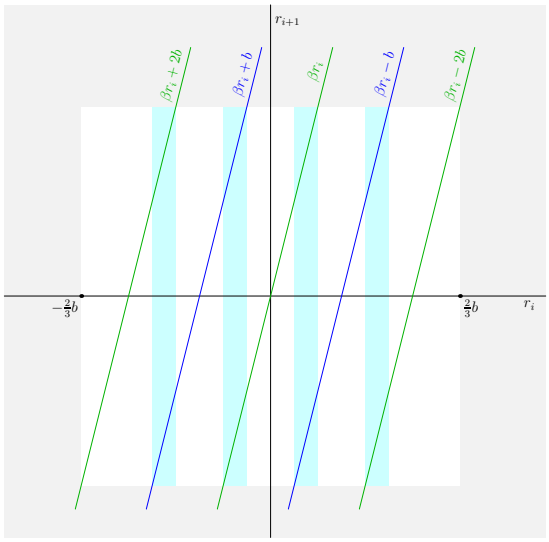
In order to get maximum redundancy in the choice of the quotient digit q, we let

$$R = \frac{\alpha}{\beta - 1}, \tag{78}$$

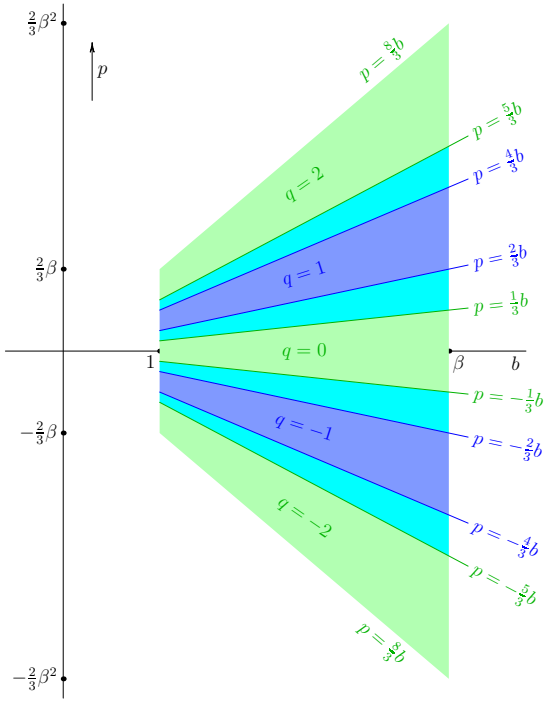
which leads to the requirement

$$\alpha \geq \frac{\beta - 1}{2}. \tag{79}$$

Figure 10 illustrates the situation $\beta = 4$ and $\alpha = 2$, thus $R = \frac{2}{3}$.



(A) Given the partial remainder r_i, the quotient digit q_i is chosen according to which of the slanted lines falls inside the white square. This ensures that the condition $-\frac{2}{3}b \leq r_i < \frac{2}{3}b$ is preserved. Note that for some values of r_i, there are two possible choices of q_i.



(B) Dependence of the quotient digit $q = q_i$ on the partial remainder $p = \beta r_i$ and the divisor b. The cyan coloured strips correspond to the pairs (b, p) with two possible choices of q.

FIGURE 10. Radix-4 SRT division.

To see that the conditions (78)-(79) are sufficient, simply set

$$q = \begin{cases} \left\lfloor \frac{\beta(r+Rb)}{b} \right\rfloor - \alpha & \text{for} \quad \beta r < (2\alpha + 1 - \beta R)b, \\ \alpha & \text{for} \quad \beta r \geq (2\alpha + 1 - \beta R)b. \end{cases} \tag{80}$$

As $r \geq -Rb$, this ensures that $-\alpha \leq q \leq \alpha$. In the first case of (80), we have

$$\beta r - qb = \beta r - b\left\lfloor \frac{\beta(r + Rb)}{b} \right\rfloor + \alpha b = \beta(r + Rb) - b\left\lfloor \frac{\beta(r + Rb)}{b} \right\rfloor - (\beta R - \alpha)b, \tag{81}$$

implying that

$$-Rb \leq \beta r - qb < (1 - R)b. \tag{82}$$

Note that $1 - R \leq \frac{1}{2} \leq R$ by (79). As for the second case of (80), a simple manipulation gives

$$(1 - R)b \leq \beta r - qb < Rb. \tag{83}$$

We conclude that (80) solves the inequality (74).

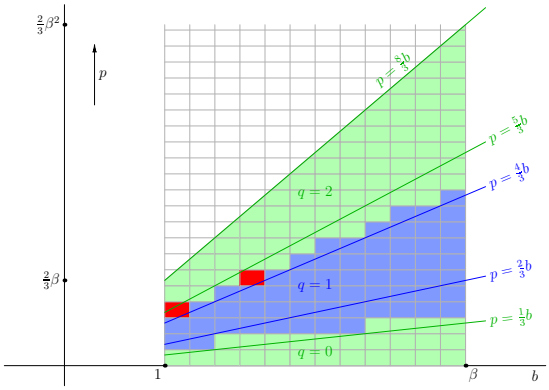
As mentioned before, the point of the SRT division is that (74) has more than one solution for some values of the partial remainder r. More precisely, assuming that

$$R \le 1, \qquad \text{or equivalently,} \qquad \alpha \le \beta - 1, \tag{84}$$

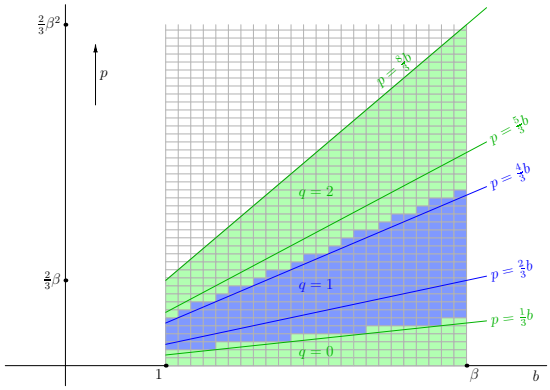
which rules out triple solutions, the inequality (74) has double solutions if and only if

$$(k+1-R)b \le \beta r < (k+R)b \qquad \text{for some} \quad k \in \{-\alpha, \dots, \alpha - 1\}. \tag{85}$$

On Figure 10, these regions are depicted in cyan.



(A) The divisor b is approximated by its leading 4 bits, while the partial remainder p is approximated by 3 bits. The two red squares show that this precision is not enough to yield a valid decision procedure for the quotient digit q.



(B) The divisor b is approximated by its leading 5 bits, while the partial remainder p is approximated by 4 bits. This precision is accurate enough to yield a valid decision procedure for the quotient digit q.

FIGURE 11. Radix-4 SRT division.

The fact that we have two equally correct choices for the quotient digit in those special regions of the (b,p)-diagram, cf. Figure 10(b), grants us the possibility to make the decision based on only a few digits of b and p. In Figure 11, we consider two attempts at designing such a decision procedure. For convenience, we depicted only the positive half of the diagram. First, we approximate the divisor b by its leading 4 bits, and the partial remainder p by 3 bits (left diagram). Here we see that there are situations (red regions) where some pairs (b,p) from the "$q=1$ only" region and some pairs (b,p) from the "$q=2$ only" region give the same combination of leading digits. One idea is to scale both b and p so that $2 \le b < 4$, which will make sure that the "problem regions" are avoided. On the other hand, by using 5 bits of b and 4 bits of p, we can design a procedure to choose a valid quotient digit (right diagram).

The main benefit of the SRT division algorithm is that since the quotient digit depends only on a few leading digits of the partial remainder, the quotient digit computation can start as soon as those digits of the partial remainder become available. The details can be organized in various ways, and we refer the reader to specialized literature on hardware arithmetic.

6. Floating point numbers

Since the real numbers are uncountable, in general, they must be approximately represented. Perhaps the simplest proposal would be to use the integers internally, but to interpret them in such a way that $m \in \mathbb{Z}$ represents κm, where κ is some fixed small constant, such as $\kappa = 0.001$. This gives us access to the subset $\kappa\tilde{\mathbb{Z}} \subset \mathbb{R}$, where $\tilde{\mathbb{Z}} \subset \mathbb{Z}$ is the set of all admissible integers in the given setting. With reference to how we represent integers, it is convenient to have $\kappa = \beta^e$, where e is a fixed integer, so that the accessible numbers are of the form

$$a = m\beta^e, \qquad m \in \tilde{\mathbb{Z}}. \tag{86}$$

For example, with $\beta = 10$ and $e = -2$, we get the numbers such as $1.00, 1.01, \dots, 1.99$, and with $\beta = 10$ and $e = -3$, we get $1.000, 1.001, \dots, 1.999$, etc. These are called *fixed point numbers*, which can be imagined as a uniformly spaced net placed on the real number line. Note that in the discussion of the division algorithms, we have implicitly used fixed point numbers.

In practice, there is a trade-off between precision and range: For example, supposing that we can store only 4 decimal digits per (unsigned) number, taking $e = -1$ gives the numbers $000.0, \dots, 999.9$, while the choice $e = -3$ would result in $0.000, \dots, 9.999$. With 64 bit integers, if we take $e = -30$ (and $\beta = 2$), which corresponds to $30 \log_{10} 2 \approx 9$ digits after the radix point in decimals, we can cover an interval of width $2^{34} \approx 17 \cdot 10^9$. Thus, fixed point numbers are only good for working with moderately sized quantities, although the implementation is straightforward and they offer a uniform precision everywhere within the range.

A simple modification of the above scheme yields a system that can handle extremely wide ranges: We let the exponent e to be a variable in (86), leading to *floating point numbers*

$$a = m\beta^e, \qquad m \in M, \quad e \in E, \tag{87}$$

where $M \subset \mathbb{Z}$ and $E \subset \mathbb{Z}$. In this context, m and e are called the *mantissa* (or *significand*) and the *exponent* of a, respectively.

- Note that the role of e is simply to tell the location of the radix point. In practice, e is moderate, so it does not require much storage. For example, the volume of the observable universe is roughly 10^{185} cubic Planck length.
- Hence the main information content of a number is in the mantissa. Thus, 1000002 and 2890.032 contain basically the same amount of information.
- The number of digits in the mantissa is called the *number of significant digits*.

Modern computers handle floating point numbers at the hardware level, following the predominant IEEE 754 standard. Perhaps the most popular among the formats provided by this standard is *double precision format*, which uses 64 bits per number as follows.

- 1 bit for the sign of m.
- 52 bits for the magnitude of m. The first bit of m is not stored here, and is taken to be 1 in the so called *normalized regime*. So the smallest positive value of m in this regime is 2^{52}, the next number is $2^{52} + 1$, and so on, the largest value is $2^{53} - 1$. In decimals, 52 bits can be rephrased as roughly 16 significant digits.
- 11 bits for e. It gives 2048 possible exponents, but 2 of these values are used as special flags, leaving 2046 possibilities, which we use as: $E = \{-1074, -1073, \dots, 971\}$.
- One of the special values of e is to signal an *underflow*, and activate the *denormalized regime*. In this regime, the first bit of m is implied to be 0, and $e = -1074$.
- Depending on the value of m, the other special value of e is used to signal signed infinities (*overflow*) or NaN (*not a number*, such as 0/0).

Example 2. To have a better understanding, let us consider a simplified model, where $\beta = 10$, two digits are allowed for m, and $E = \{-1, 0, 1\}$. In the normalized regime, the mantissa must satisfy $10 \le m \le 99$, and hence the smallest positive number is $10 \cdot 10^{-1} = 1$, and the largest number is $99 \cdot 10^1 = 990$. In the denormalized regime, the nonnegative numbers are $0, 0.1, \dots, 0.9$. This situation is illustrated in Figure 12.

As for the double precision format, the smallest normalized positive number is

$$a_* = 2^{52} \cdot 2^{-1074} = 2^{-1022} \approx 10^{-308}, \tag{88}$$

and the largest possible number is

$$a^* = (2^{53} - 1) \cdot 2^{971} \approx 2^{1024} \approx 10^{308}. \tag{89}$$

If the result of a computation goes beyond a^* in absolute value, we get an *overflow*. On the other hand, an *underflow* occurs when a computation produces a number that is smaller than

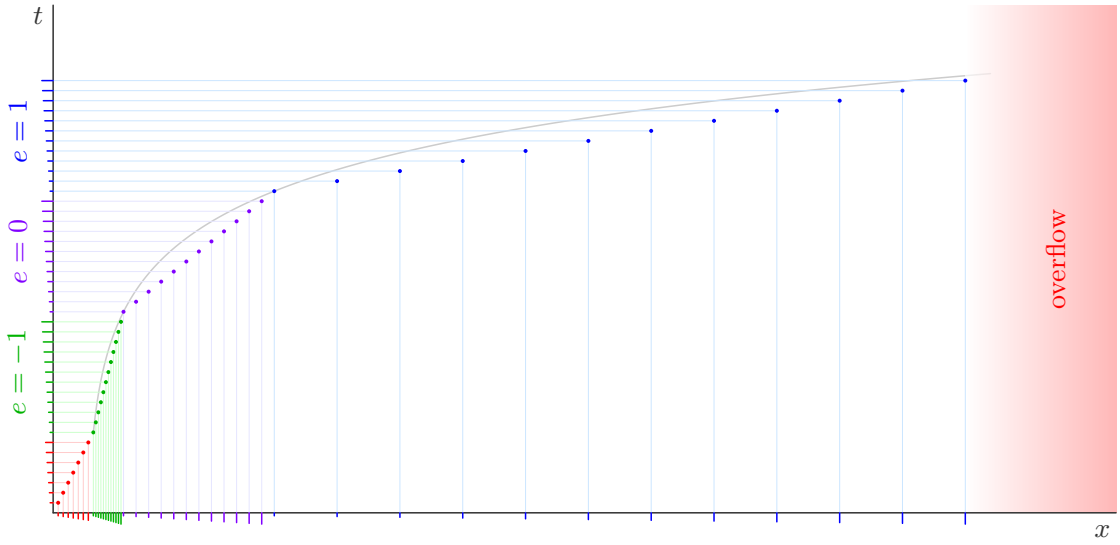


FIGURE 12. The red dots at the left signify the denormalized regime (the underflow gap). The graph of a logarithm function is shown in the background.

a_* in absolute value. In contrast to older formats where 0 was the only admissible number in the so-called underflow gap $(-a_*, a_*)$, the current standard supports *gradual underflow*, meaning that it has denormalized numbers sprinkled throughout this gap. However, gradual or not, an underflow means a contaminated outcome, and should be avoided at all cost.

Furthermore, the distance between two consecutive double precision numbers behaves like

$$\delta x \sim \varepsilon |x|, \qquad \text{where} \quad \varepsilon \approx 10^{-16}, \tag{90}$$

in the normalized regime, and

$$\delta x \sim 2^{-1022} \varepsilon, \tag{91}$$

in the denormalized regime. This can be thought of as the "resolution" of the double precision floating point numbers.

Remark 3. It is clear that the normalized regime is where we want to be in the course of any computation, and one needs a theoretical guarantee that the algorithm stays within this regime. What is the same, we need to analyze the potential situations where underflow or overflow (or NaN for that matter) could be produced, and should modify the algorithm to account for those. Once this is taken care of, the upper and lower limits of e become irrelevant, and we can set $E = \mathbb{Z}$ in (87), leading to what can be called *generalized floating point numbers*

$$\tilde{\mathbb{R}} = \tilde{\mathbb{R}}(\beta, N) = \Big\{ \pm \sum_{k=0}^{N} a_k \beta^{k+e} : 0 \le a_k \le \beta - 1,\ e \in \mathbb{Z} \Big\}. \tag{92}$$

Note that in order to ensure uniqueness of a representation $a = \pm \sum_{k=0}^{N} a_k \beta^{k+e}$ for any given $a \in \tilde{\mathbb{R}}$, we can assume $a_N \ne 0$ unless $a = 0$.

Thus in the double precision format, or in any other reasonable setting, the floating point numbers can be modelled by a set $\tilde{\mathbb{R}} \subset \mathbb{R}$, satisfying the following assumption.

Axiom 0. *There exist* $\varepsilon \ge 0$ *and a map* $\mathrm{fl} : \mathbb{R} \to \tilde{\mathbb{R}}$ *such that*

$$|\mathrm{fl}(x) - x| \le \varepsilon |x| \qquad \textit{for all} \quad x \in \mathbb{R}. \tag{93}$$

The parameter ε is called the machine epsilon *or the* machine precision.

Note that in the context of (92), we may take $\mathrm{fl}(x) = \max\{y \in \tilde{\mathbb{R}} : y \le x\}$ and $\varepsilon = \beta^{-N}$, or more precisely, $\varepsilon = (\beta-1)\beta^{-N-1}$. By taking $\mathrm{fl} : \mathbb{R} \to \tilde{\mathbb{R}}$ as rounding to the closest floating point number, we can even get $\varepsilon = \frac{1}{2}\beta^{-N}$, but this does not give any improvement for $\beta = 2$.

7. Floating point arithmetic

Let us now discuss the basic arithmetic operations on $\tilde{\mathbb{R}}$. An immediate observation is that $\tilde{\mathbb{R}}$ is in general *not* closed under arithmetic operations. For instance, thinking of $\beta = 10$ and $N = 2$ in (92), we have $1.01 \in \tilde{\mathbb{R}}$, but $1.01 \times 1.01 = 1.0201 \notin \tilde{\mathbb{R}}$. Therefore, we need to *approximate* these operations. As a benchmark, we may consider $\mathrm{fl}(x+y)$, $\mathrm{fl}(x-y)$, $\mathrm{fl}(x\times y)$, etc., as approximations to $x+y$, $x-y$, $x\times y$, etc., respectively, for which we have the estimates

$$|\mathrm{fl}(x \pm y) - (x \pm y)| \le \varepsilon |x \pm y|, \qquad |\mathrm{fl}(x \times y) - (x \times y)| \le \varepsilon |x \times y|, \qquad \text{etc.} \tag{94}$$

In practice, however, it would be inefficient to compute $x \pm y$ exactly, in order to produce $\mathrm{fl}(x \pm y)$, if x and y have very different magnitudes, as in, e.g., $10^{50} + 10^{-50}$. Hence we need a more direct way to approximate $x \pm y$.

Example 4. It turns out that a simple truncation is enough when the signs of the two summands are the same. Thus thinking of $N = 2$ and $\beta = 10$ in (92), in order to compute $101 + 2.6$, we truncate 2.6 to 2, and use $\tilde{s} = 101 + 2$ as an approximate sum.

Lemma 5 (Truncated sum)**.** *Let $\tilde{\mathbb{R}}$ be as in (92), and suppose that $a, b \in \tilde{\mathbb{R}}$ are given by*

$$a = \sum_{k=0}^{N} a_k \beta^{k+e}, \qquad \text{and} \qquad b = \sum_{k=0}^{N} b_k \beta^{k+e-m}, \tag{95}$$

with $a_N \neq 0$ and $m \ge 0$. Then the "truncated sum"

$$\tilde{s} = a + \sum_{k=m}^{N} b_k \beta^{k+e-m}, \tag{96}$$

satisfies the error bound

$$|\tilde{s} - (a+b)| \le \beta^{-N}(a+b). \tag{97}$$

Proof. First of all, note that we can set $e = 0$ since we are only interested in relative errors. Then we proceed as

$$\begin{aligned} 0 \le a + b - \tilde{s} &= \sum_{k=0}^{m-1} b_k \beta^{k-m} \le \sum_{k=0}^{m-1} (\beta-1)\beta^{k-m} \\ &\le (\beta-1)(1 + \beta + \ldots + \beta^{m-1})\beta^{-m} \\ &= (\beta^m - 1)\beta^{-m} \le 1, \end{aligned} \tag{98}$$

where we have used the fact that $b_k \le \beta - 1$. On the other hand, we have

$$a + b \ge a \ge a_N \beta^N \ge \beta^N, \qquad \text{or} \qquad 1 \le \beta^{-N}(a+b), \tag{99}$$

which completes the proof. □

Example 6. The situation with subtraction (in the sense that the summands have differing signs) is slightly more complicated. To illustrate, consider the subtraction $10.1 - 9.95 = 0.15$, and again thinking of $N = 2$ and $\beta = 10$ in (92), note that the truncated subtraction gives $10.1 - 9.9 = 0.2$. Then the absolute error is $0.2 - 0.15 = 0.05$, and the relative error is $\frac{0.05}{0.15} = \frac{1}{3}$. This is much larger than the desired level 10^{-2}.

Nevertheless, the solution is very simple: We use a few extra "guard digits" to perform subtraction, and then round the result to a nearest floating point number. In fact, a single guard digit is always sufficient. So for the preceding example, with 1 guard digit, we get $10.1 - 9.95 = 0.15$ as the intermediate result, and since $0.15 \in \tilde{\mathbb{R}}$, subtraction is exact. If we were computing, say, $15.1 - 0.666$, with 1 guard digit, the intermediate result would be $15.10 - 0.66 = 14.44$, and after rounding, our final result would be 14.4.

In the following lemma, we take $\tilde{\mathbb{R}}$ as in (92), and let $\operatorname{rnd} : \mathbb{R} \to \tilde{\mathbb{R}}$ be an operation satisfying

$$|x - \operatorname{rnd}(x)| \leq \delta \beta^{-N}, \qquad x \in \mathbb{R}, \tag{100}$$

with some constant $\frac{1}{2} \leq \delta \leq \frac{\beta-1}{\beta}$. Note that if this operation is rounding to a nearest floating point number, then $\delta = \frac{1}{2}$, whereas for a simple truncation we can set $\delta = \frac{\beta-1}{\beta}$.

Lemma 7 (Subtraction with guard digits)**.** *Let*

$$a = \sum_{k=0}^{N} a_k \beta^{k+e}, \qquad \text{and} \qquad b = \sum_{k=0}^{N} b_k \beta^{k+e-m}, \tag{101}$$

with $a_N \neq 0$ and $m \geq 0$. Then the truncated subtration with $j \geq 1$ guard digits

$$\tilde{d} = \operatorname{rnd}\Big(a - \sum_{k=m-j}^{N} b_k \beta^{k+e-m}\Big), \tag{102}$$

satisfies the error bound

$$|\tilde{d} - (a-b)| \leq \big(\delta + (1+\delta)\beta^{1-j}\big)\beta^{-N}(a-b). \tag{103}$$

In particular, taking $j = 1$ and $\delta = \frac{1}{2}$, we can ensure $|\tilde{d} - (a-b)| \leq 2\beta^{-N}(a-b)$.

Proof. Without loss of generality, assume $e = 0$. Let

$$d^* = a - \sum_{k=m-j}^{N} b_k \beta^{k+e-m}, \tag{104}$$

be the intermediate result, and note that the final result $\tilde{d} = \operatorname{rnd}(d^*)$ satisfies

$$|\tilde{d} - d^*| \leq \delta \beta^{-N} d^*. \tag{105}$$

Since the intermediate result is exact (meaning $d^* = a - b$) if $m \leq j$, we can further assume that $m > j \geq 1$. Then we proceed similarly to the proof of Lemma 5, and get

$$\begin{aligned} 0 \leq d^* - (a-b) &= \sum_{k=0}^{m-j-1} b_k \beta^{k-m} \leq \sum_{k=0}^{m-j-1} (\beta-1)\beta^{k-m} \\ &\leq (\beta-1)(1 + \beta + \ldots + \beta^{m-j-1})\beta^{-m} \\ &= (\beta^{m-j} - 1)\beta^{-m} \leq \beta^{-j}. \end{aligned} \tag{106}$$

A lower bound on $a - b$ can be obtained as follows.

$$\begin{aligned} a - b &\geq a_N \beta^N - \sum_{k=0}^{N} b_k \beta^{k-m} \geq \beta^N - \sum_{k=0}^{N} (\beta-1)\beta^{k-m} \\ &= \beta^N - (\beta-1)(1 + \beta + \ldots + \beta^N)\beta^{-m} \\ &= \beta^N - (\beta^{N+1} - 1)\beta^{-m} \geq \beta^N - \beta^{N-1} + \beta^{-m} \\ &\geq (\beta-1)\beta^{N-1}. \end{aligned} \tag{107}$$

Finally, an application of the triangle inequality, in combination with (105) and (106), gives

$$\begin{aligned} |\tilde{d} - (a-b)| &\le |\tilde{d} - d^*| + |d^* - (a-b)| \le \delta\beta^{-N} d^* + \beta^{-j} \\ &\le \delta\beta^{-N}(\beta^{-j} + a - b) + \beta^{-j} \\ &\le \delta\beta^{-N}(a-b) + \frac{(1+\delta\beta^{-N})\beta^{1-j}}{(\beta-1)}\beta^{-N}(a-b), \end{aligned} \tag{108}$$

where we have used $d^* \le \beta^{-j} + a - b$ in the penultimate step, and $1 \le \frac{\beta^{1-N}}{\beta-1}(a-b)$ from (107) in the last step. The proof is complete, as $\beta \ge 2$ and $N \ge 0$. □

Turning to multiplication and division, recall that

$$\begin{aligned} (m_1\beta^{e_1}) \times (m_2\beta^{e_2}) &= (m_1 m_2)\beta^{e_1+e_2}, \\ (m_1\beta^{e_1})/(m_2\beta^{e_2}) &= (m_1/m_2)\beta^{e_1-e_2}. \end{aligned} \tag{109}$$

To compute $m_1 m_2$ exactly, one should be able to work with $\sim 2N$ significant digits. The exact result can then be rounded to N digits. A more efficient choice would be a truncated multiplication (also called a "short product"), cf. Exercise 2. For division, we may simply apply the long division algorithm until the first N digits are obtained. Hence multiplication and division of floating point numbers are completely straightforward to implement, provided that we have good algorithms for integer arithmetic.

These discussions justify the following general assumption.

Axiom 1. *For each $\star \in \{+,-,\times,/\}$, there exists a binary operation $\circledast : \tilde{\mathbb{R}} \times \tilde{\mathbb{R}} \to \tilde{\mathbb{R}}$ such that*

$$|x \star y - x \circledast y| \le \varepsilon |x \star y|, \qquad x, y \in \tilde{\mathbb{R}}, \tag{110}$$

where of course, division by zero is excluded.

Remark 8. Once we have formulated the axioms, the idea is obviously to use them as a foundation, so that analysis (as well as design) of algorithms do not depend on the specific details of how floating point numbers were implemented. For instance, we do not need to be concerned with the parameters β and N, or even with what exactly the set $\tilde{\mathbb{R}}$ is. All we need is the knowledge that there is a set $\tilde{\mathbb{R}} \subset \mathbb{R}$ satisfying the axioms with some parameter $\varepsilon \ge 0$. In this regard, any 5-tuple $(\tilde{\mathbb{R}}, \oplus, \ominus, \otimes, \oslash)$ satisfying Axiom 0 and Axiom 1 may be called a *floating point system with machine precision* ε. Then the special case with $\tilde{\mathbb{R}} = \mathbb{R}$ and $\varepsilon = 0$ would be called *exact arithmetic*.

Remark 9. In case one needs more precision than allowed by the default floating point numbers, a robust option is *arbitrary precision* formats, which are usually implemented at the software level. Arbitrary precision simply means that the mantissa of a number is now bignum, and the arithmetic operations can be performed to stay within any given error tolerance. The cost of operations must then depend on the number of significant digits, as in Figure 1.

Exercise 2 (Short product)**.** We have reduced multiplication of floating point numbers to multiplication of two positive integers, cf (109). Recall from Section 2 the multiplication algorithm based on the Cauchy product

$$ab = \Big(\sum_{j=0}^{N} a_j\beta^j\Big) \cdot \Big(\sum_{i=0}^{N} b_i\beta^i\Big) = \sum_{k=0}^{\infty}\Big(\sum_{j=0}^{k} a_j b_{k-j}\Big)\beta^k, \tag{111}$$

cf. (8). We assume the normalization $a_N \ne 0$ and $b_N \ne 0$. With the intent of saving resources, let us ignore the terms with $k < m$ in the latter sum, with the truncation parameter m, that is, we replace the product ab by

$$\tilde{p} = \sum_{k=m}^{\infty}\Big(\sum_{j=0}^{k} a_j b_{k-j}\Big)\beta^k.$$

Show that

$$0 \le ab - \tilde{p} \le ab \cdot \beta^{m+3-2N}.$$

What would be a good choice for the value of m, in the context of floating point multiplication?

Exercise 3 (Sterbenz lemma)**.** With $\tilde{\mathbb{R}} = \tilde{\mathbb{R}}(\beta, N)$ as in (92), let $a, b \in \tilde{\mathbb{R}}$ be positive numbers satisfying $\frac{1}{2}a \le b \le 2a$. Then show that $a \ominus b = a - b$.

8. Propagation of error

A *numerical algorithm* is an algorithm that takes a finite sequence of floating point numbers (and possibly integers) as input, and produces a finite sequence of floating point numbers (and possibly integers) as output. Here a floating point number means an element of $\tilde{\mathbb{R}}$ as in the axioms. The algorithm itself can contain the usual logical constructs such as conditional statements and loops, and a set of predefined operations on integers and floating point numbers, including the arithmetic operations, comparisons, and evaluation of some elementary functions. If we fix any particular input value, then after unrolling the loops, and replacing the conditional statements by the taken branches, the algorithm becomes a simple linear sequence of operations. This sequence in general depends on details of the floating point system $(\tilde{\mathbb{R}}, \oplus, \ominus, \otimes, \oslash)$ that are more fine-grained than the axioms (i.e., on how the system is really implemented), but the idea is that we should avoid algorithms whose performances critically depend on those details, so that the axioms provide a solid foundation for all analysis. Hence in the end, we are led to the analysis of sequences such as

$$\begin{aligned} &\mathbb{R}^3 \xrightarrow{(1,1,\exp)} \mathbb{R}^3 \xrightarrow{(\times,1)} \mathbb{R}^2 \xrightarrow{+} \mathbb{R}, \\ &\tilde{\mathbb{R}}^3 \xrightarrow{(1,1,\widetilde{\exp})} \mathbb{R}^3 \xrightarrow{(\otimes,1)} \tilde{\mathbb{R}}^2 \xrightarrow{\oplus} \tilde{\mathbb{R}}, \end{aligned} \tag{112}$$

where the upper row corresponds to exact arithmetic, and the lower row to inexact arithmetic. To clarify, the preceding sequence can be rewritten as

$$(x, y, z) \mapsto (x, y, \exp z) \mapsto (xy, \exp z) \mapsto xy + \exp z, \tag{113}$$

and so it approximates the function $f(x, y, z) = xy + \exp z$ by $\tilde{f}(x, y, z) = (x \otimes y) \oplus \widetilde{\exp}(z)$, with $\widetilde{\exp} : \tilde{\mathbb{R}} \to \tilde{\mathbb{R}}$ being some approximation of exp. In the context of numerical algorithms, theoretical analysis of perturbations in the output due to the inexactness of floating point arithmetic is known as *roundoff error analysis*. We illustrate it by the example in (113).

- If all operations except the last step were exact, then we would be computing $a \oplus b$, which is an approximation of $a + b$, where $a = xy$ and $b = \exp(z)$.
- However, those operations are inexact, so the input to the last step is not the "true" (or "intended") values a and b, but their approximations $\tilde{a} = x \otimes y$ and $\tilde{b} = \widetilde{\exp}(z)$.
- Hence the computed value $\tilde{a} \oplus \tilde{b}$ will be an approximation of $\tilde{a} + \tilde{b}$.

We can put it in the form of a diagram.

$$\begin{array}{ccccc} (a, b) & \rightsquigarrow & (\tilde{a}, \tilde{b}) & & \\ \downarrow{\scriptstyle +} & & \downarrow{\scriptstyle +} & \searrow^{\oplus} & \\ a + b & \rightsquigarrow & \tilde{a} + \tilde{b} & \rightsquigarrow & \tilde{a} \oplus \tilde{b} \end{array} \tag{114}$$

Here the squiggly arrows indicate perturbations from the "true values," due to, e.g., inexact arithmetic. The error committed in the lower right squiggly arrow can be accounted for with the help of Axiom 1:

$$|\tilde{a} \oplus \tilde{b} - (\tilde{a} + \tilde{b})| \le \varepsilon |\tilde{a} + \tilde{b}|, \tag{115}$$

or equivalently,

$$\tilde{a} \oplus \tilde{b} = (1+\eta)(\tilde{a}+\tilde{b}) \qquad \text{for some} \quad |\eta| \le \varepsilon. \tag{116}$$

On the other hand, the behaviour of the error committed in the lower left squiggly arrow is something intrinsic to the operation of summation itself, since this simply reflects how the sum behaves with respect to inexact summands. Thus, putting

$$\tilde{a} = a + \Delta a, \qquad \tilde{b} = b + \Delta b, \tag{117}$$

we have

$$\tilde{a} + \tilde{b} - (a+b) = \Delta a + \Delta b, \tag{118}$$

where, e.g., $\Delta a = \tilde{a} - a$ is called the *absolute error* in $\tilde{a}$. Recall that we have access to the approximation $\tilde{a}$, but do not have access to the "true value" a. We may read (118) as: Absolute errors are simply combined during summation.

Next, dividing (118) through by $a+b$, we get

$$\varepsilon_{a+b} := \frac{\tilde{a}+\tilde{b}-(a+b)}{a+b} = \frac{a\varepsilon_a + b\varepsilon_b}{a+b}, \tag{119}$$

where, e.g., $\varepsilon_a = \frac{\Delta a}{a}$ is called the *relative error* in $\tilde{a}$. We see that relative errors get combined, with weights $\frac{a}{a+b}$ and $\frac{b}{a+b}$, respectively. In particular, if $a+b \approx 0$, then the relative error ε_{a+b} can be large, potentially catastrophic.

Remark 10. In the floating point context, the aforementioned phenomenon of potentially catastrophic growth in relative error is called *cancellation of digits*. For example, consider

$$\begin{array}{r} 1\,2\,6.1 \\ -\;1\,2\,5.8 \\ \hline 0.3 \end{array}$$

If we suppose that 126.1 and 125.8 had errors of size ≈ 0.1 in them, meaning that all their digits were significant, the error in the result 0.3 can be as large as ≈ 0.2, which is barely a 1 significant digit accuracy. Since the true result could be as small as $\approx 126.0 - 125.9 = 0.1$, the relative error of the result can only be bounded by $\approx 200\%$. The origin of the term "cancellation of digits" is also apparent: The first 3 digits of the two numbers in the input cancelled each other. We should stress here that the root cause of this phenomenon is the intrinsic sensitivity of the sum (or rather, subtraction) with respect to perturbations in the summands. It has nothing to do with the floating point arithmetic per se (In fact the subtraction in the preceding example was exact). However, cancellation of digits is a constant enemy of numerical algorithms, precisely because all inputs are inexact as they are in most cases the result of an inexact operation.

Turning back to the (119), assuming that $|\varepsilon_a| \le \varepsilon$ and $|\varepsilon_b| \le \varepsilon$, we get

$$|\varepsilon_{a+b}| \le \frac{|a|+|b|}{|a+b|}\,\varepsilon. \tag{120}$$

Here, we can think of the quantity

$$\kappa_+(a,b) = \frac{|a|+|b|}{|a+b|}, \tag{121}$$

as expressing the sensitivity of $a+b$ with respect to perturbations in a and b. This is called the *condition number of addition*.

We proceed further, by using the triangle inequality and the estimate (115), as

$$|\tilde{a} \oplus \tilde{b} - (a+b)| \le |\tilde{a} \oplus \tilde{b} - (\tilde{a}+\tilde{b})| + |\tilde{a}+\tilde{b}-(a+b)| \le \varepsilon|\tilde{a}+\tilde{b}| + \varepsilon_{a+b}|a+b|, \tag{122}$$

where we are now thinking of ε_{a+b} as a manifestly nonnegative quantity (i.e., we denoted $|\varepsilon_{a+b}|$ by ε_{a+b}). Then invoking

$$|\tilde{a}+\tilde{b}| \leq (1+\varepsilon_{a+b})|a+b|, \tag{123}$$

and (120), we end up with

$$\frac{|\tilde{a}\oplus\tilde{b}-(a+b)|}{|a+b|} \leq \varepsilon(1+\varepsilon_{a+b})+\varepsilon_{a+b} \leq \big((1+\varepsilon)\kappa_+(a,b)+1\big)\varepsilon, \tag{124}$$

which takes into account both inexactness of the input, and inexactness of the summation operation.

Let us do the same analysis for multiplication. We start with

$$\tilde{a}\tilde{b}-ab = b\Delta a + a\Delta b + \Delta a\Delta b, \tag{125}$$

and division by ab yields

$$\varepsilon_{ab} := \frac{\tilde{a}\tilde{b}-ab}{ab} = \varepsilon_a+\varepsilon_b+\varepsilon_a\varepsilon_b \approx \varepsilon_a+\varepsilon_b. \tag{126}$$

Thus, relative errors are simply combined during multiplication. If we assume that $|\varepsilon_a| \leq \varepsilon$ and $|\varepsilon_b| \leq \varepsilon$, we get

$$|\varepsilon_{ab}| \leq 2\varepsilon+\varepsilon^2 \leq \kappa_\times(a,b)\cdot\varepsilon, \tag{127}$$

where the *condition number of multiplication* is

$$\kappa_\times(a,b) \approx 2. \tag{128}$$

The full analysis involving inexact multiplication is exactly the same as in the case of addition, and the final result we obtain is

$$\frac{|\tilde{a}\otimes\tilde{b}-ab|}{|ab|} \leq \varepsilon(1+\varepsilon_{ab})+\varepsilon_{ab} \leq \big((1+\varepsilon)\kappa_\times(a,b)+1\big)\varepsilon. \tag{129}$$

Quantitatively, of course $\kappa_\times(a,b)$ remains bounded independently of a and b, while $\kappa_+(a,b)$ can become unbounded, exhibiting cancellation of digits.

Remark 11 (Univariate functions)**.** Let $f : I \to \mathbb{R}$ be a differentiable function, with $I \subset \mathbb{R}$ being an open interval. Suppose that $\tilde{x} = x+\Delta x$ is a perturbation of the "true value" $x \in I$, and let $z = f(x)$. Then we have

$$\tilde{z} := f(\tilde{x}) = f(x+\Delta x) \approx f(x)+f'(x)\Delta x, \tag{130}$$

and so

$$\Delta z := \tilde{z}-z \approx f'(x)\Delta x, \tag{131}$$

for Δx small. From this, we can estimate the relative error, as

$$\frac{\Delta z}{z} \approx \frac{f'(x)\Delta x}{f(x)} = \frac{xf'(x)}{f(x)}\cdot\frac{\Delta x}{x}. \tag{132}$$

The quantity

$$\kappa_f(x) = \frac{xf'(x)}{f(x)} = \frac{(\log f(x))'}{(\log x)'}, \tag{133}$$

is called the *(asymptotic) condition number of f at x*, which represents the relative error amplification factor, in the asymptotic regime where the error is small. The sign of κ_f has a little importance, so we are really thinking of having the absolute value in the right hand side of (133). Note that this can be thought of as the "derivative measured against relative error," or as the derivative of $\log f$ taken with respect to the variable $\log x$. Since the argument of a function always involves perturbation, either due to measurement error, initial rounding, or inexact operations in the preparation steps, the condition number reflects the *intrinsic difficulty* of computing the function in floating point arithmetic.

Example 12. Let us compute the condition numbers for some common functions.

- For $f(x) = \frac{1}{x}$, we have $\kappa(x) = 1$.
- For $f(x) = x^\alpha$, we have $\kappa(x) = \alpha$.
- For $f(x) = e^x$, we have $\kappa(x) = x$.
- For $f(x) = x - 1$, we have $\kappa(x) = \frac{x}{x-1} = 1 + \frac{1}{x-1}$. Cancellation of digits at $x = 1$.
- For $f(x) = x + 1$, we have $\kappa(x) = \frac{x}{x+1}$. Cancellation of digits at $x = -1$.
- For $f(x) = \cos x$, we have $\kappa(x) = x \tan x$. Cancellation of digits at $x = \frac{\pi}{2} + \pi n$, $n \in \mathbb{Z}$.

Example 13. Consider the root $x = 1 - \sqrt{1-q}$ of the quadratic $x^2 - 2x + q = 0$, where we assume $q \approx 0$. Suppose that $\sqrt{1-q}$ was computed with relative error ε, i.e., the computed root is $\tilde{x} = 1 - (1+\varepsilon)\sqrt{1-q}$. Then we have

$$\frac{x - \tilde{x}}{x} = \frac{\varepsilon\sqrt{1-q}}{x} \approx \frac{2\varepsilon}{q}, \tag{134}$$

where we have used the fact that $x \approx \frac{q}{2}$ for $q \approx 0$. Since $\frac{2\varepsilon}{q} \to \infty$ as $q \to 0$, our algorithm exhibits cancellation of digits. This occurs even if the input argument q is its true value, because the computation of $\sqrt{1-q}$ is inexact. We may think of the algorithm as decomposing the function $f(q) = 1 - \sqrt{1-q}$ into two factors, as

$$f = g \circ h, \tag{135}$$

where $g(y) = 1 - y$ and $h(q) = \sqrt{1-q}$. Since $h(q)$ is computed inexactly, and $g(y)$ is poorly conditioned near $y = 1$, we get cancellation of digits.

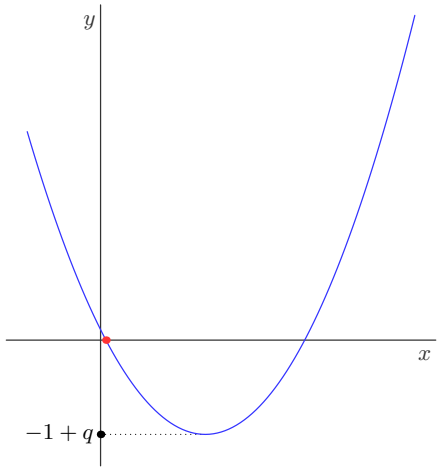


FIGURE 13. The quadratic $y = x^2 - 2x + q$.

Can this be fixed? To answer this question, we compute the condition number of f:

$$\kappa_f(q) = \frac{qf'(q)}{f(q)} = \frac{q}{2\sqrt{1-q}(1-\sqrt{1-q})} \approx 1, \qquad \text{for} \quad q \approx 0, \tag{136}$$

which indicates that there should be no intrinsic difficulty of computing $f(q)$ for $q \approx 0$, and hence the preceding algorithm, i.e., the decomposition (135), was a poorly designed one. In fact, a way out suggests itself, if we keep in mind that subtraction of nearly equal quantities should be avoided. Namely, the following transformation gives a well behaved algorithm.

$$f(q) = 1 - \sqrt{1-q} = (1 - \sqrt{1-q})\frac{1+\sqrt{1-q}}{1+\sqrt{1-q}} = \frac{q}{1+\sqrt{1-q}}. \tag{137}$$

Let us go over the involved operations one by one. First, since $q \approx 0$, the subtraction $1 - q$ is well conditioned. Second, the square root is always well conditioned for positive numbers, cf, Example 12. Then as $\sqrt{1-q} \approx 1$, the summation $1 + \sqrt{1-q}$ is well conditioned. Finally, division is always well conditioned. We conclude that the algorithm implicitly defined by (137) does not have cancellation of digits near $q = 0$.

Exercise 4. Perform a detailed roundoff error analysis of (137).

Remark 14 (Bivariate functions)**.** Let $f : U \to \mathbb{R}$ be a continuously differentiable bivariate function, with $U \subset \mathbb{R}^2$ being an open set. Suppose that $\tilde{x} = x + \Delta x$ and $\tilde{y} = y + \Delta y$ are perturbations of the "true values" $(x, y) \in U$, and let $z = f(x, y)$. Then we have

$$\begin{aligned} \tilde{z} := f(\tilde{x}, \tilde{y}) &\approx f(\tilde{x}, y) + \frac{\partial f}{\partial y}(\tilde{x}, y)\Delta y \approx f(x, y) + \partial_x f(x, y)\Delta x + \partial_y f(\tilde{x}, y)\Delta y \\ &\approx f(x, y) + \partial_x f(x, y)\Delta x + \partial_y f(x, y)\Delta y, \end{aligned} \tag{138}$$

where in the last step we have used the continuity of $\partial_y f$. This gives

$$\frac{\tilde{z} - z}{z} \approx \frac{\partial_x f(x, y)}{f(x, y)}\Delta x + \frac{\partial_y f(x, y)}{f(x, y)}\Delta y = \frac{x\partial_x f(x, y)}{f(x, y)} \cdot \frac{\Delta x}{x} + \frac{y\partial_y f(x, y)}{f(x, y)} \cdot \frac{\Delta y}{y}, \tag{139}$$

and assuming that the relative errors of $\tilde{x}$ and $\tilde{y}$ are of the same magnitude, we are led to the definition that the asymptotic condition number of f is

$$\kappa_f(x, y) = \left|\frac{x\partial_x f}{f}\right| + \left|\frac{y\partial_y f}{f}\right| = \kappa_{x,f} + \kappa_{y,f}, \tag{140}$$

which can be thought of as the sum of two condition numbers, one in x direction and the other in y direction.

Example 15. Let us compute the conditions numbers for some common functions.

- For $f(x, y) = x + y$, we have $\kappa(x, y) = \frac{|x|+|y|}{|x+y|}$, cf. (121).
- For $f(x, y) = x^\alpha y^\beta$, we have $\kappa(x, y) = |\alpha| + |\beta|$. Putting $\alpha = 1$ and $\beta = -1$, we get the (asymptotic) condition number of division.

Exercise 5. Generalize the notion of condition number to functions of n variables. Then compute the condition numbers of the sum and the product of n numbers.

Exercise 6. Let $I \subset \mathbb{R}$ and $J \subset \mathbb{R}$ be open intervals, thought of as the domain and codomain of some collection of functions $f : I \to J$. We associate to I and J the "error metrics"

$$e(x, \Delta x) = g(x)\Delta x, \qquad e(z, \Delta z) = h(z)\Delta z, \tag{141}$$

where, e.g., $e(x, \Delta x)$ is to be understood as the error measure of the perturbation Δx near the point x, and g and h are positive functions. For a differentiable function $f : I \to J$, we have $\Delta z \approx f'(x)\Delta x$, and so

$$h(z)\Delta z \approx h(z)f'(x)\Delta x = \frac{xh(f(x))f'(x)}{g(x)} \cdot g(x)\Delta x, \tag{142}$$

leading us to define the *generalized asymptotic condition number*

$$\kappa_f(x) = \frac{xh(f(x))f'(x)}{g(x)}, \tag{143}$$

associated to the error metrics (141). Note that the usual condition number (133) is obtained by setting $g(x) = \frac{1}{x}$ and $h(z) = \frac{1}{z}$.

(a) Take $I = J = \mathbb{R}$, and find all error metrics for which the generalized condition number of any translation $f(x) = x + a$ ($a \in \mathbb{R}$) is equal to 1.

(b) Take $I = J = (0, \infty)$, and find all error metrics for which the generalized condition number of any scaling $f(x) = \lambda x$ $(\lambda > 0)$ is equal to 1.

9. Summation and product

In this section, we consider computation of the sum and product

$$s_n = x_1 + x_2 + \ldots + x_n, \qquad p_n = x_1 \times x_2 \times \ldots \times x_n, \tag{144}$$

of a given collection $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$. First of all, let us look at the condition numbers. Thus, introduce perturbations

$$\tilde{x}_k = x_k + \Delta x_k, \qquad k = 1, \ldots, n, \tag{145}$$

and let

$$\tilde{s}_n = \tilde{x}_1 + \tilde{x}_2 + \ldots + \tilde{x}_n, \qquad \tilde{p}_n = \tilde{x}_1 \times \tilde{x}_2 \times \ldots \times \tilde{x}_n. \tag{146}$$

Then we have

$$\tilde{s}_n - s_n = \Delta x_1 + \Delta x_2 + \ldots + \Delta x_n, \tag{147}$$

and assuming that $|\Delta x_k| \le \varepsilon |x_k|$ for $k = 1, \ldots, n$, we get

$$\frac{|\tilde{s}_n - s_n|}{|s_n|} \le \frac{|x_1| + |x_2| + \ldots + |x_n|}{|s_n|}\, \varepsilon. \tag{148}$$

From this, we read off the *condition number of summation*

$$\kappa_+(x) = \frac{|x_1| + |x_2| + \ldots + |x_n|}{|x_1 + x_2 + \ldots + x_n|}, \tag{149}$$

which is a generalization of (121). Naturally, we have cancellation of digits near $s_n = 0$.

Turning to product, we start with

$$\begin{aligned} \tilde{p}_n - p_n &= \tilde{x}_1\tilde{x}_2\cdots\tilde{x}_n - x_1x_2\cdots x_n = (\tilde{x}_1 - x_1)\tilde{x}_2\cdots\tilde{x}_n + x_1\tilde{x}_2\cdots\tilde{x}_n - x_1x_2\cdots x_n \\ &= (\tilde{x}_1 - x_1)\tilde{x}_2\cdots\tilde{x}_n + x_1(\tilde{x}_2 - x_2)\tilde{x}_3\cdots\tilde{x}_n + x_1x_2\tilde{x}_3\cdots\tilde{x}_n - x_1x_2\cdots x_n = \ldots \\ &= (\tilde{x}_1 - x_1)\tilde{x}_2\cdots\tilde{x}_n + x_1(\tilde{x}_2 - x_2)\tilde{x}_3\cdots\tilde{x}_n + \ldots + x_1x_2\cdots x_{n-1}(\tilde{x}_n - x_n). \end{aligned} \tag{150}$$

Then invoking the estimates $|\tilde{x}_k - x_k| \le \varepsilon |x_k|$ and $|\tilde{x}_k| \le (1 + \varepsilon)|x_k|$, we infer

$$\begin{aligned} |\tilde{p}_n - p_n| &\le \big(\varepsilon(1+\varepsilon)^{n-1} + \varepsilon(1+\varepsilon)^{n-2} + \ldots + \varepsilon(1+\varepsilon) + \varepsilon\big)|x_1x_2\cdots x_n| \\ &= \big((1+\varepsilon)^n - 1\big)|p_n| = \Big(n\varepsilon + \binom{n}{2}\varepsilon^2 + \ldots + n\varepsilon^{n-1} + \varepsilon^n\Big)|p_n| \\ &\le \big(n\varepsilon + n^2\varepsilon^2 + \ldots + n^{n-1}\varepsilon^{n-1} + n^n\varepsilon^n\big)|p_n| \\ &\le \frac{n\varepsilon}{1 - n\varepsilon}|p_n|, \end{aligned} \tag{151}$$

where we have assumed that $n\varepsilon < 1$. This implies that for perturbations satisfying, say, $n\varepsilon \le \frac{1}{2}$, the *condition number of product* satisfies

$$\kappa_\times(x) \le 2n, \tag{152}$$

and asymptotically, we have $\kappa_\times(x) \le n$ as $\varepsilon \to 0$.

Next, we look at the effect of inexact arithmetic on products with unperturbed input. Introduce the notation

$$\bar{p}_k = x_1 \otimes x_2 \otimes \ldots \otimes x_k, \qquad k = 1, 2, \ldots, n, \tag{153}$$

and invoking Axiom 1, we have

$$\begin{aligned}
\bar{p}_2 &= x_1 \otimes x_2 = (1+\eta_1)x_1x_2,\\
\bar{p}_3 &= \bar{p}_2 \otimes x_3 = (1+\eta_2)p_2x_3 = (1+\eta_1)(1+\eta_2)x_1x_2x_3,\\
&\dots\\
\bar{p}_n &= \bar{p}_{n-1} \otimes x_n = (1+\eta_{n-1})p_{n-1}x_n = (1+\eta_1)(1+\eta_2)\cdots(1+\eta_{n-1})x_1x_2\cdots x_n,
\end{aligned} \tag{154}$$

for some $\eta_1, \dots, \eta_{n-1}$ satisfying $|\eta_k| \le \varepsilon$, $k = 1, \dots, n-1$. This yields

$$|\bar{p}_n - p_n| = |(1+\eta_1)(1+\eta_2)\cdots(1+\eta_{n-1}) - 1||p_n| \le \big((1+\varepsilon)^{n-1} - 1\big)|p_n| \tag{155}$$

and so

$$\frac{|\bar{p}_n - p_n|}{|p_n|} \le \frac{(n-1)\varepsilon}{1-(n-1)\varepsilon} \le 2n\varepsilon, \tag{156}$$

where in the last step we have assumed that $n\varepsilon \le \frac{1}{2}$.

Exercise 7. Combine the effects of input perturbation and inexact arithmetic for products. That is, estimate $\tilde{x}_1 \otimes \dots \otimes \tilde{x}_n - x_1 \times \dots \times x_n$, where the notations are as above.

Finally, we deal with inexact summation.

Example 16 (Swamping)**.** Thinking of $N = 2$ and $\beta = 10$ in (92), we have $10.0 \oplus 0.01 = 10.0$, and by repeating this operation, we get, for instance

$$10.0 \oplus \underbrace{0.01 \oplus 0.01 \oplus \dots \oplus 0.01}_{100\,\text{times}} = 10.0, \tag{157}$$

giving the relative error $\approx 10\%$. On the other hand, we have

$$\underbrace{0.01 \oplus 0.01 \oplus \dots \oplus 0.01}_{100\,\text{times}} \oplus 10.0 = 11.0, \tag{158}$$

which is the exact result. This suggests that one should sum the small numbers first. In particular, floating point addition is *not associative*.

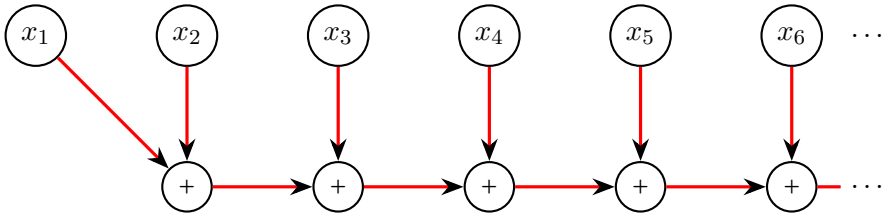


FIGURE 14. The "naive" summation algorithm (159). Errors made in the early steps get amplified more, as they must go through all the subsequent steps.

As with the product, introduce the notation

$$\bar{s}_k = x_1 \oplus x_2 \oplus \dots \oplus x_k, \qquad k = 1, 2, \dots, n, \tag{159}$$

and invoking Axiom 1, we have

$$\begin{aligned}
\bar{s}_2 &= x_1 \oplus x_2 = (1+\eta_1)(x_1 + x_2),\\
\bar{s}_3 &= \bar{s}_2 \oplus x_3 = (1+\eta_2)(p_2 + x_3) = (1+\eta_1)(1+\eta_2)(x_1+x_2) + (1+\eta_2)x_3,\\
&\dots\\
\bar{s}_n &= \bar{s}_{n-1} \oplus x_n = (1+\eta_{n-1})(s_{n-1} + x_n)\\
&= (1+\eta_1)\cdots(1+\eta_{n-1})(x_1+x_2) + (1+\eta_2)\cdots(1+\eta_{n-1})x_3 + \dots + (1+\eta_{n-1})x_n,
\end{aligned} \tag{160}$$

for some $\eta_1, \dots, \eta_{n-1}$ satisfying $|\eta_k| \le \varepsilon$, $k = 1, \dots, n-1$. This yields

$$\begin{aligned} |\bar{s}_n - s_n| &\le |(1+\eta_1)\cdots(1+\eta_{n-1}) - 1||x_1 + x_2| \\ &\quad + |(1+\eta_2)\cdots(1+\eta_{n-1}) - 1||x_3| + \ldots + |\eta_{n-1}||x_n| \\ &\le \big((1+\varepsilon)^{n-1} - 1\big)|x_1 + x_2| + \big((1+\varepsilon)^{n-2} - 1\big)|x_3| + \ldots + \varepsilon|x_n|. \end{aligned} \tag{161}$$

Since $(1+\varepsilon)^k - 1 = k\varepsilon + O(k^2\varepsilon^2)$, terms such as x_1 and x_2 carry more weight in the final error than terms such as x_n, explaining the swamping phenomenon we have seen in Example 16. By using the simple estimate $(1+\varepsilon)^k - 1 \le \frac{n\varepsilon}{1-n\varepsilon} =: \rho(\varepsilon, n)$ on all pre-factors, we arrive at

$$\frac{|\bar{s}_n - s_n|}{|s_n|} \le \frac{n\varepsilon}{1-n\varepsilon} \cdot \frac{|x_1| + \ldots + |x_n|}{|x_1 + \ldots + x_n|} = \rho(\varepsilon, n)\kappa_+(x) \le 2n\kappa_+(x)\varepsilon, \tag{162}$$

where in the last step we have assumed that $n\varepsilon \le \frac{1}{2}$.

Remark 17. Recall that the condition number $\kappa_+(x)$ reflects how error propagates through the summation map $x \mapsto s$. Then $\rho(n, \varepsilon)$ has to do with the particular way this map is implemented, i.e., how $x \mapsto s$ is approximated by a sequence of floating point operations.

Exercise 8. Combine the effects of input perturbation and inexact arithmetic for summation. That is, estimate $\tilde{x}_1 \oplus \ldots \oplus \tilde{x}_n - (x_1 + \ldots + x_n)$, where the notations are as above.

Exercise 9. To reduce the parameter $\rho(n, \varepsilon) = O(n\varepsilon)$, as well as to alleviate the phenomenon of swamping, one may consider the *pairwise summation* algorithm, depicted in Figure 15.

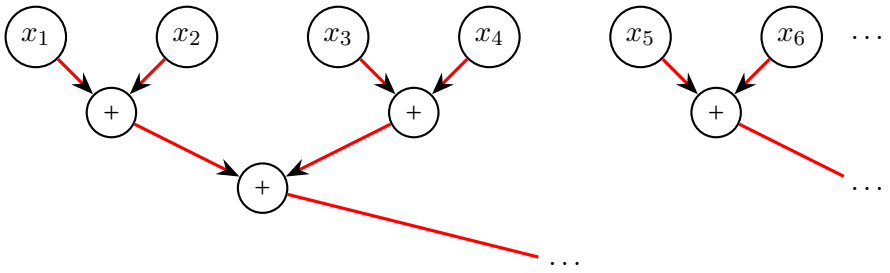


FIGURE 15. Pairwise summation algorithm.

More precisely, we set

$$\sigma(x_1, x_2) = x_1 \oplus x_2, \tag{163}$$

and

$$\sigma(x_1, x_2, \dots, x_{2k}) = \sigma(x_1, x_2, \dots, x_k) \oplus \sigma(x_{k+1}, x_{k+2}, \dots, x_{2k}), \tag{164}$$

for $k \ge 2$. This defines an algorithm for summing $x_1, x_2, \dots, x_n$, when n is a power of 2.

(a) Extend the algorithm to arbitrary integer n, not necessarily a power of 2.
(b) Show that $\rho(n, \varepsilon) = O(\varepsilon \log n)$ for this algorithm.

Exercise 10. Let $x_1, x_2, \dots$ be a sequence of floating point numbers, and let $s_n = x_1 + \ldots + x_n$. Consider *Kahan's compensated summation algorithm*

$$\begin{aligned} y_n &= x_n + e_{n-1} \\ \tilde{s}_n &= \tilde{s}_{n-1} + y_n \\ e_n &= (\tilde{s}_{n-1} - \tilde{s}_n) + y_n, \qquad n = 1, 2, \dots \end{aligned}$$

where each operation is performed in floating point arithmetic, and $\tilde{s}_0 = e_0 = 0$.

(a) Explain why you would expect the roundoff accuracy of this method to be better than that of the naive summation method.

(b) Show that

$$|\tilde{s}_n - s_n| \leq [C\varepsilon + O(\varepsilon^2)] \sum_{k=1}^{n} |x_k|,$$

where C is some constant, and ε is the machine epsilon.