Exact Complex Arithmetic in an Imaginary Radix System

Alexander Kaganovsky

14 July, 1999

Abstract

This paper investigates an exact arithmetic based on the single-component representation of complex numbers by sequences of signed digits written to imaginary base ri. Algorithms for the four basic arithmetic operations in this representation are described and analyzed. The algorithms are to an unexpected extent scarcely different from their exact real equivalents, which significantly speeds up exact complex number manipulations.

1 Introduction

The complex numbers originated from the desire for a symbolic representation for the solutions of such equations as $x^2 + 1 = 0$, otherwise irreducible over $\mathbb{R}$. In modern terminology, we say that the field $\mathbb{C}$ of complex numbers is a finite algebraic extension of $\mathbb{R}$ of degree 2, $\mathbb{C} = \mathbb{R}\left(\sqrt{-1}\right)$, which by mere coincidence also happens to be the *algebraic closure* of $\mathbb{R}$ (see e.g. [1]). The latter fact is precisely the famous Fundamental Theorem of Algebra, and the exception rather than the rule. As well as algebraically closed, $\mathbb{C}$ also turns out to be *complete* with respect to the norm that extends the norm $|\cdot|$ on $\mathbb{R}$, which makes it the convenient number system, as it is, in which to study calculus and analysis.

Historically speaking, the definition of the complex numbers came before the rigorous definition of the real numbers in terms of Cauchy sequences. In an apparent attempt to put things in logical order, it became conventional to discuss complex numbers in terms of the real number pairs used to represent them. Following this mathematical tradition, in most modern digital computers, complex numbers are also represented as pairs of real numbers, and arithmetic operations on complex numbers are developed in terms of the corresponding operations on reals. The need to maintain separate representations for the real and imaginary parts of complex numbers makes them much more awkward to compute than the reals. For instance, complex addition or subtraction involves two real additions or subtractions, while multiplication of two complex numbers involves four real multiplications, a real addition, and a real subtraction:

$$(a + ib)(c + id) = (ac - bd) + i(bc + ad).$$

If multiplication is a much slower operation than addition, the above formula can be improved upon by using the relation

$$(a + ib)(c + id) = (ac - bd) + i[(a + b)(c + d) - ac - bd],$$

which involves only three real multiplications (ac, bd, $(a + c)(b + d)$), two real additions and three real subtractions [2]. Complex division in this representation, however, is inevitably even more "complex":

$$\frac{a + ib}{c + id} = \frac{ac + bd}{c^2 + d^2} + i\frac{bc - ad}{c^2 + d^2}.$$

In this report, we introduce a method of performing *infinite precision* arithmetic on complex numbers that does not involve separating the complex numbers into their real and imaginary parts. This is achieved by using the positional representation of complex numbers by lazy infinite sequences of real digits written to a complex base. We show that the four basic arithmetic operations in this notation can be performed similarly to exact real arithmetic in base r^2, with the exception of the normalization procedure.

1.1 Infinite precision arithmetic

It is well-known that the use of floating-point arithmetic for real number computations can sometimes lead to results that are highly inaccurate or even completely meaningless. The following sequence (due to Muller [3]):

$$x_{k+1} = 111 - (1130 - 3000/x_{k-1})/x_k, \quad x_0 = 11/2, \quad x_1 = 61/11,$$

provides an unsettling glimpse of such round-off anomalies, which persist even on multiple precision: in exact arithmetic x_n is a monotonically increasing sequence that converges to 6, yet in floating point it rapidly converges to 100 after only 14 divisions and 12 subtractions. Represented as pairs of reals, complex numbers inherit all round-off, over- and under-flow misbehaviours of real floating-point arithmetic, and matters become even worse as round-off errors can obscure the singularities of complex analytic functions [4].

However, lest one's faith in computer arithmetic be completely undermined, a number of alternatives to floating-point arithmetic have been suggested, the main three being rational arithmetic, interval arithmetic and *infinite precision* or *exact real* arithmetic. The first two methods are not entirely satisfactory: interval arithmetic does not produce exact results and tends to be overly pessimistic in measuring errors, while rational arithmetic is not closed under many operations of interest (e.g., square root), and eventually becomes very slow and inefficient, as an extended sequence of computations may result in very large numerators and denominators.

Exact real arithmetic, on the other hand, represents numbers as *infinite* objects, so that mathematical operations on the set of number representations faithfully represent the corresponding operations on real numbers. The chief approaches to exact real arithmetic include the functional approach [5, 6], the lazy sequence approach, formulated in terms of both redundant signed-digit sequences [5, 6, 7] and continued fractions [8], and also a recent blend of the two [9]. The forthcoming discussion of redundant radix-ri signed-digit sequences is also somewhat in parallel with my earlier paper [10] on exact real arithmetic.

1.2 Unified representation of complex numbers

On account of the complexity of even the four basic arithmetic operations, several proposals have been made, some as early as in 1960's, for a more concise, single-component representation of the complex numbers, which allow complex arithmetic to be done in a unified manner, relieving the programmer of the burden of treating the real and imaginary parts separately [11, 12, 13, 14]. The essence of these proposals is to choose the radix to be a complex or an imaginary number, and then use sequences of *real* digits to express the complex quantity as a weighted sum of powers of that radix. The proposals have so far been only applied to finite systems.

In this paper, we investigate the use of the number ri as the radix of an infinite positional weighted system. For this system, we assume a positional notation

$$z_0.z_1z_2\cdots z_n\cdots = \sum_{k=0}^{\infty} z_k \cdot (ri)^{-k},$$

where the weights associated with position $k \in \mathbb{N}_0$ are $(ri)^{-k}$:

$$\begin{array}{cccccccccc} k & : & 0 & 1 & 2 & 3 & 4 & 5 & 6 & \cdots \\ (ri)^{-k} & : & 1 & -r^{-1}i & -r^{-2} & r^{-3}i & r^{-4} & -r^{-5}i & -r^{-6} & \cdots \end{array}$$

As can be seen from the above, the even-numbered positions in this expansion all have real weights, while the odd-numbered positions all have imaginary weights. The real weights are related to each other as in *negative* radix $(-r^2)$, and so are the imaginary ones after factoring out the common ri (see Figures 1, 2).

$$\begin{array}{ccccccc} k & : & 0 & 2 & 4 & 6 & \cdots \\ (ri)^{-k} & : & 1 & -r^{-2} & r^{-4} & -r^{-6} & \cdots \end{array}$$

Figure 1: Even-numbered positions

$$\begin{array}{cccccccc} k & : & 1 & 3 & 5 & \cdots & \\ (ri)^{-k} & : & -r^{-2} & r^{-4} & -r^{-6} & \cdots & \times ri \end{array}$$

Figure 2: Odd-numbered positions (after factoring out ri)

For example, let $r = 10$ and the number z be represented in radix $10i$ by the following expansion:

$$z = 19\,.\,36\ 23\ 40$$

Then

$$\begin{aligned} z &= 19 - 36 \cdot 10^{-1}\, i - 23 \cdot 10^{-2} + 40 \cdot 10^{-3}\, i \\ &= 18.77 - 3.56\, i \end{aligned}$$

Thus, in order to express a complex number $z = x + iy$ in this format, we first write its real part x as an expansion in radix negative r^2, then we write a radix $(-r^2)$ expansion of ry, and finally we interleave the two parts to obtain a single-component expansion.

Negative-radix systems have been extensively described in the literature [15, 16, 17, 18], and the rules for addition, negation, multiplication and division of finite numbers in these systems can be obtained by almost straightforward modification of the standard rules. At least one computer has been built based on such a system [19]. However, the only advantage of negative-radix number systems seems to reside in the fact that all finite real numbers, whether positive or negative, can be represented in these systems without a separate sign digit, thus obviating the need for special treatment of negative numbers [16].

1.3 Redundant signed-digit representations

Even though the idea of absorbing the sign into the number representation may have a moderate practical advantage in fixed-size-format number systems, where only a finite precision is available for representing numbers, the changeover from finite to infinite renders this method indistinguishable from the conventional positive-radix method.

Indeed, it is well-known that in order for even the four basic operations of arithmetic to be computable on-line in a radix-r number system, the latter has to exhibit redundancy [20, 7]. The most commonly used radix-r redundant number system is the so-called *symmetric signed-digit* system, based on a digit set

$$S_\rho = \left\{\overline{\rho}, \ldots, \overline{1}, 0, 1, \ldots, \rho\right\},$$

where $\overline{x}$ denotes $-x$, $1 \le \rho \le r-1$ and $\rho \ge r/2$. What makes the system redundant is the last condition, which allows each digit to assume more than r values.

The use of a symmetrical signed-digit set blurs the distinction between positive- and negative-radix number systems and allows us to circumvent the difficulties of coping with alternating signs in negative-radix expansions. Indeed, if we have a signed-digit expansion in a radix $r > 0$,

$$x = \sum_{k=0}^{\infty} x_k r^{-k}, \ -\rho \le x_k \le \rho,$$

we can easily transform it into the corresponding negative-radix expansion

$$x = \sum_{k=0}^{\infty} x'_k \, (-r)^{-k} \, , \ -\rho \le x'_k \le \rho,$$

where

$$x'_k = (-1)^k \, x_k, \ k \in \mathbb{N}_0 \, ,$$

and vice versa.

Now let us consider a signed-digit radix-ri expansion

$$\sum_{k=0}^{\infty} z_k \cdot (ri)^{-k} \tag{1}$$

of a complex number $z = x + iy$ with

$$z_k \in \mathbb{Z}, \ |z_k| \le \rho, \ \rho \in \left[\frac{r^2}{2}, \, r^2 - 1\right]. \tag{2}$$

It is clear that only a subset of the complex numbers can be represented by such sequences; namely, the numbers $z = x + iy$ with

$$|x| \le \rho \frac{r^2}{r^2 - 1}, \ |y| \le \rho \frac{r}{r^2 - 1}. \tag{3}$$

Note that the representation (1) of a complex number $z = x + iy$ is asymmetrical with respect to x and y because of the presence of the factor $1/r$ in the expansion of y:

$$\begin{aligned} x &= z_0 - z_2 r^{-2} + z_4 r^{-4} + \dots \, , \\ y &= \frac{1}{r} \left(-z_1 + z_3 r^{-2} - z_5 r^{-4} + \dots \right). \end{aligned} \tag{4}$$

This brings up the question: How can we represent *all* complex numbers in the form (1)? The standard solution to this problem is to *scale* any complex number onto the above range (3). The scaling could in principle be applied separately to the real and imaginary parts of the complex number, but the arithmetic operations become much simpler when the real and imaginary parts share a *common* exponent [21].

1.4 Exponent representation

The exponent representation of complex numbers is similar to the commonly used floating-point representation of real numbers and consists of two main parts: the exponent $E \in \mathbb{Z}$, and the mantissa M, which is a sequence of integers $(z_n)_{n=0}^{\infty} \in \mathbb{Z}$. . The complex number z, represented by the pair (E, M), has the value

$$z = B^E \cdot \sum_{n=0}^{\infty} z_n \, (ri)^{-n} \,, \tag{5}$$

where B is the *base* of the exponent (usually a positive integer).

It is easy to show that *all* complex numbers can be represented in a radix-ri system as an expansion of the form (5), where all z_n are in the range (2). Note that for the time being, we are interested primarily in the general properties of this system, and do not concern ourselves with the questions of computability or effective convergence of the series in (5). Indeed, let $Z \in \mathbb{C}$ be an arbitrary complex number, $Z = X + iY$, $X, Y \in \mathbb{R}$. We aim to find an exponent $E \in \mathbb{Z}$ and two numbers $x, y \in \mathbb{R}$ satisfying condition (3) such that

$$Z = B^E \cdot (x + iy) \,. \tag{6}$$

From (6) and (3), we find that

$$B^E \geq \frac{r^2 - 1}{\rho r^2} \, |X| \,, \; B^E \geq \frac{r^2 - 1}{\rho r} \, |Y| \,,$$

and therefore,

$$B^E \geq \max \left(\frac{r^2 - 1}{\rho r^2} \, |X| \,, \, \frac{r^2 - 1}{\rho r} \, |Y| \right) . \tag{7}$$

The best value of E in (7) is the minimal one, because it corresponds to the largest possible values of x and y in (6) that satisfy (3). Thus, we choose the exponent

$$E = \left\lceil \log_B \max \left(\frac{r^2 - 1}{\rho r^2} \, |X| \,, \, \frac{r^2 - 1}{\rho r} \, |Y| \right) \right\rceil .$$

2 Redundant radix-ri exponent representation of complex numbers

The main decision to be made regarding the exponent representation is the choice of base B. The choice partially depends on the way numbers are generated in the system, and the usual practice is to choose the exponent base to be the same as the main radix. We can assume that most of the complex data will be fed into the system in the conventional form, with the real and imaginary parts separately represented in real radix r', where r' is either r or r^2. Leaving negative bases such as $(-r)$ or $(-r^2)$ out of consideration, we have only two practicable choices for the exponent base: r and r^2. In either case, the real number pairs first have to be converted to $(-r^2)$, and then interleaved to obtain a radix-ri expansion (conversion from and to radix-ri is discussed in Section 4). The particular choice depends on how the trade-off between addition and conversion is to be made — as we shall see, exponent base r simplifies conversion at the expense of addition, while r^2 streamlines addition to the detriment of conversion. We naturally regard addition as more important than conversion; after all, conversion should only be performed twice — once at the beginning and once at the end of a computation. Taking the aforesaid into account, we have chosen r^2 to be the exponent base.

Definition 1 *We define a computable exact complex number z as a quadruple (R, ρ, E, M), where $R \in \mathbb{N}$, $R > 4$ is a perfect square ($R = r^2$), and the base of the signed exponent $E \in \mathbb{Z}$, the range parameter ρ is an integer with $R/2 < \rho < R - 1$, and the mantissa M is an effectively given sequence of numbers $(z_n)_{n\in\mathbb{N}_0} \in \mathbb{Z}$ such that*

$$|z_n| \leq Cn, \;\; n \in \mathbb{N}, \tag{8}$$

where $C > 0$ is a constant common to all complex numbers in a system. The value of $z = (R, \rho, E, (z_n)_{n\in\mathbb{N}_0})$ is taken as

$$z = R^E \cdot \sum_{n=0}^{\infty} z_n (ri)^{-n}.$$

The convergence criterion (8) is somewhat arbitrary and only required to ensure effective convergence of the sequence; an error message could be produced at run-time if the sequence were found to violate (8).

If the mantissa entries

$$z_0, z_1, z_2, \ldots, z_n, \ldots$$

are digits in radix r^2 in the range $[-\rho, \rho]$, they represent mantissa in the domain

$$\left[-\rho\frac{r^2}{r^2-1}, \rho\frac{r^2}{r^2-1}\right] \times \left[-\rho\frac{r}{r^2-1}, \rho\frac{r}{r^2-1}\right], \tag{9}$$

and the corresponding number (R, ρ, E, M) is said to be *normalized.*

For brevity and ease of reading, we shall not always distinguish between a number z, its representation (R, ρ, E, M), and its expansion (5). Unless otherwise stated, we shall also assume that r and ρ are constant and sometimes use the notation (E, M) instead of (R, ρ, E, M).

3 Basic Arithmetic Operations

In this section, we introduce the algorithms for carrying out addition, subtraction, multiplication and division of exact complex numbers represented by infinite sequences of signed digits in radix ri. The algorithms are unexpectedly similar to their real counterparts — where they differ is the normalization procedure. This leads us to believe that the algorithms for the basic arithmetic operations remain unchanged in any radix, whether a real or complex number, as long as there is a way of normalizing sequences of digits written to that radix.

3.1 Normalization

Normalization, in the context of radix-r redundant signed-digit positional weighted systems, refers to the process of restoring the individual digits of an effectively given sequence $(a_n)_{n\in\mathbb{N}_0} \in \mathbb{Z}$, to the canonical range $[-\rho, \rho]$, where ρ is an integral range parameter between $r/2$ and $r-1$. By this is meant the finding of a *normalized* sequence, $(a'_n)_{n\in\mathbb{N}} \in [-\rho, \rho]$ such that

$$\sum_{n=0}^{\infty} a'_n r^{-n} = \sum_{n=0}^{\infty} a_n r^{-n}.$$

Note that the first digit, a'_0, of thus normalized sequence may generally remain unbounded.

Similarly, a *canonical* representation for the mantissa of a *complex* number z written to base ri is a sequence

$$(z_0, z_1, z_2, \cdots, z_n, \cdots)_{ri},$$

where $|z_n| < \rho$ for all $n \in \mathbb{N}_0$, except perhaps $n = 0$ and 1 (ρ is now a positive integer between $R/2$ and $R-1$, $R = r^2$).

Let $(z_i)_{i\in\mathbb{N}_0}$ be an unnormalized mantissa of a complex number $z = r^E \cdot \sum_{i=0}^{\infty} z_i (ri)^{-i}$. Recalling that

$$\sum_{i=0}^{\infty} z_i (ri)^{-i} = \sum_{k=0}^{\infty} (-1)^k z_{2k} R^{-k} + i \cdot \frac{1}{r} \sum_{k=0}^{\infty} (-1)^{k+1} z_{2k+1} R^{-k},$$

the problem of ri-normalizing $(z_n)_{n\in\mathbb{N}_0}$ is tantamount to normalizing two radix-R sequences $(x_n)_{n\in\mathbb{N}_0}$ and $(y_n)_{n\in\mathbb{N}_0}$, where

$$x_n = (-1)^n z_{2n}, \quad y_n = (-1)^{n+1} z_{2n+1}. \tag{10}$$

Once the sequences $(x_n)_{n\in\mathbb{N}_0}$ and $(y_n)_{n\in\mathbb{N}_0}$ have been normalized in radix R, we can once again alternate their signs in accordance with (10), and combine them into back into a single sequence z_n.

Normalization of radix-R sequences has been discussed in great detail in [10] and is summarized below.

3.1.1 Normalization of radix-R sequences

Let us first consider a sequence $(a_n)_{n\in\mathbb{N}_0}$ with $|a_n| \le R+\rho-1$, $n \in \mathbb{N}_0$, and show how to obtain a new sequence $(a'_n)_{n\in\mathbb{N}_0}$ such that

$$\sum_{n=0}^{\infty} a'_n R^{-n} = \sum_{n=0}^{\infty} a_n R^{-n}, \tag{11}$$

and

$$|a'_n| \le \rho, \ n \in \mathbb{N}. \tag{12}$$

To achieve this, we first repeatedly divide a_n by R for all $n \in \mathbb{N}_0$:

$$a_n = d_n R + m_n, \quad |m_n| < R, \quad \operatorname{sgn}(m_n) = \operatorname{sgn}(d_n).$$

We then have:

$$\begin{aligned}\sum_{n=0}^{\infty} a_n R^{-n} = \sum_{n=0}^{\infty} (d_n R + m_n) R^{-n} &= \sum_{n=0}^{\infty} d_n R^{-n+1} + \sum_{n=0}^{\infty} m_n R^{-n} \\ &= (a_0 + d_1) + \sum_{n=1}^{\infty} (m_n + d_{n+1}) R^{-n}. \end{aligned} \tag{13}$$

The condition $|a_n| \le R+\rho-1$ implies $|d_n| \le 1$ and $|m_n| \le R-1$ for all $n \in \mathbb{N}_0$; thus,

$$|a_0 + d_1| \le |a_0| + 1, \quad |m_n + d_{n+1}| \le R.$$

The sequence

$$(a_0 + d_1, m_1 + d_2, m_2 + d_3, \cdots)$$

is almost the one we are looking for, except its elements are bounded in absolute value by R, instead of ρ. To find a sequence bounded by ρ, we slightly change the parameters d_n and m_n:

$$d'_n = \begin{cases} d_n, & \text{if } |m_n| < \rho \\ d_n + \operatorname{sgn}(m_n), & \text{if } |m_n| \geq \rho \end{cases}$$
$$m'_n = \begin{cases} m_n, & \text{if } |m_n| < \rho \\ m_n - \operatorname{sgn}(m_n) \cdot R, & \text{if } |m_n| \geq \rho \end{cases} \tag{14}$$

Now the sequence

$$a'_0 = a_0 + d'_1, \quad a'_n = m'_n + d'_{n+1} \ (n \in \mathbb{N}) \tag{15}$$

satisfies both normalization conditions (11) and (12).

To verify that the first condition holds true, we note that $a_n = d'_n R + m'_n$, and similarly to (13), arrive at

$$\sum_{n=0}^{\infty} a'_n R^{-n} = (a_0 + d'_1) + \sum_{n=1}^{\infty} \left(m'_n + d'_{n+1}\right) R^{-n} = \sum_{n=0}^{\infty} a_n R^{-n}.$$

To verify the second normalization condition, we note that $|d'_n| \leq 1$ for all $n \in \mathbb{N}_0$, so if $|m_n| < \rho$, then $m'_n = m_n$, $|m'_n| < \rho$, and $|a'_n| = \left|m'_n + d'_{n+1}\right| \leq \rho$, while if $|m_n| \geq \rho$, then $m'_n = m_n - \operatorname{sgn}(m_n) \cdot R$, $1 \leq |m'_n| \leq R - \rho$, and $|a'_n| = \left|m'_n + d'_{n+1}\right| \leq R - \rho + 1$. Now since we require $r \geq 2$, or $R \geq 4$, we have

$$R - \rho + 1 \leq \rho.$$

Thus we conclude that

$$a_0 - 1 \leq a'_0 \leq a_0 + 1, \quad |a'_n| \leq \rho, \ n \in \mathbb{N}.$$

In this manner, we have constructed a function $f : \mathbb{Z}^2 \times \mathbb{Z}^{\mathbb{N}_0} \to \mathbb{Z}^{\mathbb{N}_0}$ — which will be hereinafter referred to as `reduce` — that assigns to any triple $\left(R, \rho, (a_n)_{n\in\mathbb{N}_0}\right)$ the sequence $(a'_i)_{i\in\mathbb{N}_0} \in \mathbb{Z}$, calculated according to formulae (14) and (15). Evaluation of this function can be performed totally in parallel (Figure 3).

Before we consider a more general case $|a_n| \leq M$, $n \in \mathbb{N}_0$, where $M > 0$ is an arbitrary positive integer, it is pertinent to note that the condition $|a_n| \leq R + \rho - 1$ was introduced for the sake of (12), and is not relevant to the definition of `reduce`. We can apply `reduce(R,ρ)` to any sequence $(a_n)_{n\in\mathbb{N}_0}$ for which the series $\sum_{n=0}^{\infty} a_n R^{-n}$ is convergent, and the resulting sequence $(a'_n)_{n\in\mathbb{N}_0}$ will still satisfy the first normalization condition (11).

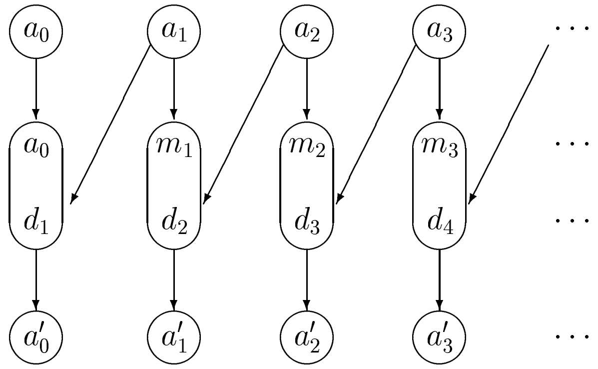

Figure 3: Totally parallel normalization

Let us show that if $|a_n| \leq M$ for all $n \in \mathbb{N}_0$, mantissa $(a_n)_{n\in\mathbb{N}_0}$ can be normalized in a finite number of steps. Indeed, applying `reduce` once, we shall obtain a sequence $(a'_n)_{n\in\mathbb{N}_0}$, satisfying the following condition:

$$|a'_n| = \left|m'_n + d'_{n+1}\right| \leq |m'_n| + \left|d'_{n+1}\right| \leq \rho - 1 + \left\lfloor \frac{M}{R} \right\rfloor + 1,$$

or

$$|a'_n| \leq M_1 \stackrel{def}{=} \left\lfloor \frac{M}{R} \right\rfloor + \rho.$$

Applying `reduce` again, we get another sequence $(a''_n)_{n\in\mathbb{N}_0}$, satisfying

$$|a''_n| \leq M_2 \stackrel{def}{=} \left\lfloor \frac{M_1}{R} \right\rfloor + \rho,$$

etc. The sequence $M, M_1, M_2, \dots$ is a sequence of decreasing natural numbers, and if $M = m_n R^n + \ldots + m_1 R + m_0$, the algorithm will terminate in at most $n+1$ steps.

More specifically, we can prove that if for some $k \in \mathbb{N}$ a sequence $(a_n)_{n\in\mathbb{N}_0}$ satisfies $|a_n| \leq R^k + \rho - 1$ for all $n \in \mathbb{N}$, it can be fully normalized by `reduce` in at most k steps. This enables us to determine the number of times k one has to apply `reduce` in order to fully normalize a given sequence $(a_n)_{n\in\mathbb{N}_0}$ with $|a_n| \leq M$, $n \in \mathbb{N}$; namely,

$$k = \min\left\{n \in \mathbb{N} \mid M \leq R^n + \rho - 1\right\}.$$

Solving the inequality $M \leq R^n + \rho - 1$ for $n \in \mathbb{N}$, we find that

$$n \geq \log_R (M - \rho + 1),$$

and, therefore,

$$k = \lceil \log_R (M - \rho + 1) \rceil. \tag{16}$$

3.1.2 Normalization of radix-ri sequences

We have shown that normalization of a radix-ri sequence is performed through normalizing its even and odd subsequences (10) in radix r^2. Once the subsequences have been r^2-normalized, all that remains for us to do is alternate their signs and combine them back into a single sequence z_n. The complex analogue of `reduce` will be referred to as `creduce`.

3.2 Addition and Subtraction

3.2.1 Addition of two complex numbers

Let $z = (E_z, (z_0, z_1, \cdots, z_n, \cdots)_{ri})$ and $w = (E_w, (w_0, w_1, \cdots, w_n, \cdots)_{ri})$ be the two normalized complex numbers to be added. If $E_z = E_w$, the procedure for addition is very straightforward: the digits of the two sequences are added and the resulting sequence

$$(z_0 + w_0,\ z_1 + w_1, \cdots,\ z_n + w_n, \cdots)_{ri}$$

is then normalized. The normalization can be done in a single pass, since

$$|z_n + w_n| \le |z_n| + |w_n| \le 2\rho \le R + \rho - 1.$$

If the exponents of z and w are not equal, the mantissa of one of the operands has to be adjusted to make them equal. The shift has to be to the right to avoid the loss of the most significant digits — therefore, it is the mantissa of the addend with the *smaller* exponent that is shifted. Since addition is a commutative operation, we can assume that $e = E_z - E_w > 0$ without loss of generality. The details of the radix alignment are as follows:

$$w = R^{E_z} \cdot (w'_0, w'_1, \ldots, w'_n, \ldots)_{ri},$$

where

$$(w'_n)_{n\in\mathbb{N}_0} = \begin{cases} (\underbrace{0, \cdots, 0}_{2e \text{ zeros}}, w_0, \cdots, w_n, \cdots)_{ri}, & \text{if } e \equiv 0 \bmod 2 \\ (\underbrace{0, \cdots, 0}_{2e \text{ zeros}}, -w_0, \cdots, -w_n, \cdots)_{ri}, & \text{if } e \equiv 1 \bmod 2 \end{cases} \tag{17}$$

The mantissa in (17) is left unchanged if the number of right shifts is divisible by 4, and negated otherwise. That such is the case is explained by the fact that two right-shifts are equivalent to dividing the mantissa by $(-r^2) = -R$, while four right-shifts are equivalent to dividing it by $r^4 = R^2$:

$$\begin{aligned} R^E \cdot (w_0, \cdots, w_n, \cdots)_{ri} &= R^{E+1} \cdot (0, 0, -w_0, \cdots, -w_n, \cdots)_{ri} \\ &= R^{E+2} \cdot (0, 0, 0, 0, w_0, \cdots, w_n, \cdots)_{ri}. \end{aligned}$$

3.2.2 Addition of several complex numbers

The above addition algorithm can be readily modified to operate with n complex numbers, where $n > 2$. The procedure is essentially the same — all n numbers are aligned to match the one with the largest exponent, their mantissas are added digit by digit, and the resulting sequence — normalized by applying `creduce` k times, where

$$k = \lceil \log_R (n\rho - \rho + 1) \rceil$$

is calculated from (16) with $M = n \cdot \rho$.

Note that this is considerably more efficient than adding the n numbers pairwise using $(n-1)$ nested additions, as shown in [10].

3.2.3 Subtraction

Subtraction of a radix-ri sequence is carried out in the regular way — by negating the minuend and adding the result to the subtrahend:

$$\left(E_z, (z_n)_{n\in\mathbb{N}_0}\right)_{ri} - \left(E_w, (w_n)_{n\in\mathbb{N}_0}\right)_{ri} = \left(E_z, (z_n)_{n\in\mathbb{N}_0}\right)_{ri} + \Big(- \left(E_w, (w_n)_{n\in\mathbb{N}_0}\right)_{ri}\Big),$$

where

$$- \left(E_w, (w_n)_{n\in\mathbb{N}_0}\right)_{ri} = \left(E_w, (-w_n)_{n\in\mathbb{N}_0}\right)_{ri}.$$

3.3 Multiplication

Complex multiplication in radix ri is also a very straightforward modification of radix-r multiplication discussed in [10], and is performed by multiplying the formal series and renormalizing the result. If $z = R^{E_z} \cdot \sum_{k=0}^{\infty} z_k (ri)^{-k}$ and $w = R^{E_w} \cdot \sum_{m=0}^{\infty} w_m (ri)^{-m}$ are the normalized numbers to be multiplied, we form their Cauchy product[1]

$$\sum_{k=0}^{\infty} u_k (ri)^{-k} = \sum_{k=0}^{\infty} \left(\sum_{m=0}^{k} z_m w_{k-m} \right) (ri)^{-k},$$

[1] The Cauchy product $\sum_{k=0}^{\infty} u_k\,(ri)^{-k}$ converges to zw as long as both series $\sum_{i=0}^{\infty} z_i\,(ri)^{-i}$ and $\sum_{j=0}^{\infty} w_j\,(ri)^{-j}$ are absolutely convergent (Mertens' theorem — see e.g. [22])

where $u_k = \left(\sum_{m=0}^{k} z_m w_{k-m} \right)$, and proceed to ri-normalize the u_k in groups of n rows each.

Specifically, if $(z_n)_{n\in\mathbb{N}_0}$ and $(w_n)_{n\in\mathbb{N}_0}$ are canonical representations of z and w, then $|z_n| \le \rho$ and $|w_n| \le \rho$, hence

$$|u_n| \le \rho^2 \cdot (n+1)\,.$$

We want to find the result in the form

$$u = zw = r^{E_u} \cdot \sum_{m=0}^{\infty} u'_m \, (ri)^{-m}\,,$$

where $|u'_m| \le \rho$ for all $m \in \mathbb{N}_0$. The sequence u_n cannot be normalized directly, because it is generally unbounded. The way out is to recursively normalize small bounded portions of $(u_m)_{m\in\mathbb{N}_0}$, as shown in Figures 4 and 5.

$$\begin{array}{cccc|ccccc|c}
z_0w_0 & z_0w_1 & \cdots & z_0w_{N-1} & z_0w_N & z_0w_{N+1} & z_0w_{N+2} & \cdots & z_0w_{2N-1} & z_0w_{2N} \\
 & z_1w_0 & \cdots & z_1w_{N-2} & z_1w_{N-1} & z_1w_N & z_1w_{N+1} & \cdots & z_1w_{2N-2} & z_1w_{2N-1} \\
 & & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
 & & & z_{N-1}w_0 & z_{N-1}w_1 & z_{N-1}w_2 & z_{N-1}w_3 & \cdots & z_{N-1}w_N & z_{N-1}w_{N+1} \\
 & & & & z_Nw_0 & z_Nw_1 & z_Nw_2 & \cdots & z_Nw_{N-1} & z_Nw_N \\ \hline
 & & & & & z_{N+1}w_0 & z_{N+1}w_1 & \cdots & z_{N+1}w_{N-2} & z_{N+1}w_{N-1} \\
 & & & & & & z_{N+2}w_0 & \cdots & z_{N+2}w_{N-3} & z_{N+2}w_{N-2} \\
 & & & & & & & \ddots & \vdots & \vdots \\
 & & & & & & & & z_{2N-1}w_0 & z_{2N-1}w_1 \\
 & & & & & & & & & z_{2N}w_0
\end{array}$$

Figure 4: Multiplication — before normalizing

$$\begin{array}{cccc|ccccc|c}
u_{00} & u_{01} & \cdots & u_{0,N-1} & u_{0N} & u_{0,N+1} & u_{0,N+2} & \cdots & u_{0,2N-1} & u_{0,2N} \\ \hline
 & & & & & z_{N+1}w_0 & z_{N+1}w_1 & \cdots & z_{N+1}w_{N-2} & z_{N+1}w_{N-1} \\
 & & & & & & z_{N+2}w_0 & \cdots & z_{N+2}w_{N-3} & z_{N+2}w_{N-2} \\
 & & & & & & & \ddots & \vdots & \vdots \\
 & & & & & & & & z_{2N-1}w_0 & z_{2N-1}w_1 \\
 & & & & & & & & & z_{2N}w_0
\end{array}$$

Figure 5: Multiplication — after normalizing first $(N+1)$ lines

Normalization of first $(N+1)$ lines in Figure 4 takes

$$k(N) = \left\lceil \log_R \left(\rho^2 (N+1) - \rho + 1 \right) \right\rceil$$

applications of `creduce` and yields a normalized sequence whose first N digits give the first N digits of the product zw (Figure 5). Adding another N lines and renormalizing in $k(N)$ applications of `creduce` produces another sequence whose first N digits are then taken to be the next N digits of the product, and the algorithm proceeds recursively with the series $\sum_{n=0}^{\infty} u_n (ri)^{-n}$, computing the result in blocks of N digits.

Although the algorithm works correctly with any value of $N \in \mathbb{N}$, our main concern is to minimize the time and space requirements. The optimal value of N normally depends on the number of required precision digits, which is generally unknown in advance. The values of N for which $k(N) = 2$ are clearly inadequate, resulting in an unnecessarily large number of operations, but any of the numbers

$$N_m = \max \{ n \in N \mid k(N) \leq m \}, \; m \geq 3$$

appear to be equally suitable for the value of N (in terms of operation counts). The m in N_m is the number of normalizations to be performed at each step of the algorithm, and minimizing normalization is important to reduce the space requirements (the larger the m the more space is required to hold the intermediate results). In our implementation, we have used $N = N_3$ (e.g. $N = N_3 = 228$ for $r = 10$ and $\rho = 51$). For a more detailed complexity analysis, see [10].

3.4 Radix-ri division

In radix-r systems, signed-digit division of real numbers is usually performed using the method originally due to Robertson [23]. The substance of the algorithm lies with an iterative process that produces one digit of the quotient per cycle according to the following recurrence equation similar to the paper-and-pencil method:

$$P_{n+1} = r \left(P_n - q_n D \right), \; n \in \mathbb{N}_0, \tag{18}$$

where $P_0 = N$, P_n is the current partial remainder, P_{n+1} is the next partial remainder, and q_n is the quotient digit inferred from P_n and D. It is easy to see that

$$P_n = r^n \left[N - \left(q_0 + q_1 r^{-1} + \ldots + q_{n-1} r^{-n+1} \right) D \right], \; n \in \mathbb{N},$$

and so imposing an upper bound on the value of $|P_n|$ ensures convergence of the algorithm, provided that selection of the quotient digit q_n results in the next partial remainder P_{n+1} adhering to the same allowed range as P_n.

The existing signed-digit division algorithms primarily differ in their selection of quotient digits, restriction of the range of the possible values of the divisor, dividend and partial remainders and, finally, normalization techniques. Usually, selection functions make a guess about q_n based on the inspection of *several most significant digits* of P_n and D. Even though this could result in some quotient digits q_n being wrong, the redundancy allows recovery from wrong guesses by taking an appropriate correction step in the next quotient digit.

If the first most significant digit d_0 of the divisor is large enough — the exact range depends on the particular division method used — the usual selection function is

$$q_n = \left\lfloor \left| \frac{p_{n0}}{d_0} \right| \right\rfloor \cdot \operatorname{sgn}\left(\frac{p_{n0}}{d_0} \right),$$

where p_{n0} is the first most significant digit of the n-th partial remainder P_n. Thus, for example, the first most significant digit of the quotient is calculated as

$$q_0 = \left\lfloor \left| \frac{n_0}{d_0} \right| \right\rfloor \cdot \operatorname{sgn}\left(\frac{n_0}{d_0} \right).$$

If $z \in \mathbb{C}$ is the dividend and $w \in \mathbb{C}$, $w \neq 0$ the divisor, the algorithm for radix-ri system can be obtained from the real one by using a recurrence relation similar to (18):

$$P_{n+1} = ri \cdot (P_n - q_n w),\ n \in \mathbb{N}_0,$$

where $P_0 = z$ and $(q_n)_{n\in\mathbb{N}_0}$ are the digits of the quotient $Q = z/w$. In perfect analogy with the real case, we infer that

$$P_n = (ri)^n \left[z - \left(q_0 + q_1 (ri)^{-1} + \ldots + q_{n-1} (ri)^{-n+1} \right) w \right],\ n \in \mathbb{N},$$

and so if Q is correct to n radix-ri digits, we have

$$\left| z - \left(q_0 + q_1 (ri)^{-1} + \ldots + q_{n-1} (ri)^{-n+1} \right) w \right| \leq \frac{|P_n|}{r^n}.$$

If q_n were selected in such a way that P_n stayed bounded, the above relation would guarantee convergence of the algorithm.

On the digit level, let z and w be represented by the following radix-ri sequences:

$$z = (z_0, z_1, \cdots, z_n, \cdots)_{ri}, \quad w = (w_0, w_1, \cdots, w_n, \cdots)_{ri},$$

where w is appropriately scaled so that either $w_0 \neq 0$ or $w_1 \neq 0$. To gain an impression of how to choose the digit selection function, we observe that

$$\begin{aligned} P_{n+1} &= ri \cdot (p_{n0} - q_n w_0, p_{n1} - q_n w_1, \cdots, p_{nk} - q_n w_k, \cdots)_{ri} \\ &= \left(p_{n1} - q_n w_1, -r^2 (p_{n0} - q_n w_0) + p_{n2} - q_n w_2, p_{n3} - q_n w_3, \cdots\right)_{ri}. \end{aligned}$$

The algorithm will converge only if the elements of P_n stay bounded for all $n \in \mathbb{N}$, which can be achieved if q_n are selected in such a way that both $p_{n1} - q_n w_1$ and $-r^2 (p_{n0} - q_n w_0) + p_{n2} - q_n w_2$ remain relatively small. It is only these two elements that may present a problem, since the rest of the sequence can be normalized in the usual way.

Remembering that

$$\frac{z}{w} = \frac{(\operatorname{Re} z)(\operatorname{Re} w) + (\operatorname{Im} z)(\operatorname{Im} w)}{(\operatorname{Re} w)^2 + (\operatorname{Im} w)^2} + i \cdot \frac{(\operatorname{Im} z)(\operatorname{Re} w) - (\operatorname{Re} z)(\operatorname{Im} w)}{(\operatorname{Re} w)^2 + (\operatorname{Im} w)^2}, \quad (19)$$

it is clear that we have to look at *two* digits at a time, and use

$$\begin{aligned} q_n &= \left\lfloor \left| \frac{p_{n0} w_0 + p_{n1} w_1 / r^2}{w_0^2 + w_1^2 / r^2} \right| \right\rfloor \cdot \operatorname{sgn}\left(\frac{p_{n0} w_0 + p_{n1} w_1 / r^2}{w_0^2 + w_1^2 / r^2} \right) \\ &= \left\lfloor \left| \frac{r^2 p_{n0} w_0 + p_{n1} w_1}{r^2 w_0^2 + w_1^2} \right| \right\rfloor \cdot \operatorname{sgn}\left(\frac{r^2 p_{n0} w_0 + p_{n1} w_1}{r^2 w_0^2 + w_1^2} \right) \end{aligned}$$

as the digit selection function. This seemingly complex formula is derived directly from (19), remembering that

$$\begin{aligned} \operatorname{Re}(z_0 + z_1 (ri)^{-1} + \ldots + z_n (ri)^{-n} + \ldots) &= z_0 - z_2 r^{-2} + z_4 r^{-4} + \ldots \\ \operatorname{Im}(z_0 + z_1 (ri)^{-1} + \ldots + z_n (ri)^{-n} + \ldots) &= \frac{1}{r} \left(-z_1 + z_3 r^{-2} - z_5 r^{-4} + \ldots \right). \end{aligned}$$

The complete division normalization procedures require more sophisticated reasoning and are beyond the scope of this paper (see e.g. [24]).

4 Conversion to and from single-component representation

As demonstrated earlier, the single-component representation of complex numbers possesses some speed advantages in executing multiplication and division. However, in order for any number system to be of interest from the computer arithmetic standpoint, there must also be an efficient algorithmic procedure for converting numbers into the new form, as well as decoding them back into conventional form. In this section, we present the complete algorithms for the *constructor function* for creating a complex number from its real and imaginary parts, and the *selector function* for extracting real or imaginary parts of complex values. Other useful functions, such as those for replacing real or imaginary parts of a complex number while leaving the other part untouched, can be worked out in the same way.

4.1 The constructor function

As mentioned above, we shall assume that the real and imaginary parts of a complex number z are given by their respective radix-r signed-digit exponent representations

$$x = r^{E_x} \cdot \sum_{n=0}^{\infty} x_n r^{-n},$$
$$y = r^{E_y} \cdot \sum_{n=0}^{\infty} y_n r^{-n},$$

where $E_x, E_y \in \mathbb{Z}$ and $|x_n|, |y_n| \leq \rho_1$ ($\rho_1 \in [r/2, r-1]$) for all $n \in \mathbb{N}_0$. In order to compute a radix-ri exponent signed-digit representation of $z = x+iy$, we have to convert both x and y to radix r^2, having appropriately aligned the radix points. More precisely, we have to find an $E \in \mathbb{Z}$ and sequences $(x'_n)_{n\in\mathbb{N}_0}$ and $(y'_n)_{n\in\mathbb{N}_0}$ such that

$$x = r^{2E} \cdot \sum_{n=0}^{\infty} x'_n \left(r^2\right)^{-n},$$
$$y = r^{2E-1} \cdot \sum_{n=0}^{\infty} y'_n \left(r^2\right)^{-n},$$

which, according to (4), will then give us

$$z = x + iy = R^E \cdot (x'_0, -y'_0, -x'_1, y'_1, x'_2, -y'_2, \dots)_{ri}. \tag{20}$$

The radix alignment is performed as follows:

1) if $E_x > E_y$ and $E_x \equiv 0 \bmod 2$, then we set

$$\begin{aligned}
E &= E_x/2 \\
x &= r^{2E} \cdot (x_0, x_1, \cdots, x_n, \cdots)_r \\
y &= r^{2E-1} \cdot \left(\underbrace{0, 0, \cdots, 0}_{E_x - E_y - 1 \text{ zeros}}, y_0, y_1, \cdots, y_n, \cdots \right)_r,
\end{aligned}$$

where $E_x - E_y - 1 = (2E-1) - E_y \geq 0$;

2) if $E_x > E_y + 1$ and $E_x \equiv 1 \bmod 2$, then

$$\begin{aligned}
E &= (E_x - 1)/2 \\
x &= r^{2E} \cdot (rx_0 + x_1, x_2, \cdots, x_n, \cdots)_r \\
y &= r^{2E-1} \cdot \left(\underbrace{0, 0, \cdots, 0}_{E_x - E_y - 2 \text{ zeros}}, y_0, y_1, \cdots, y_n, \cdots \right)_r,
\end{aligned}$$

where $(2E-1) - E_y = E_x - E_y - 2 \geq 0$;

3) if $E_x = E_y + 1$ and $E_x \equiv 1 \bmod 2$ (and therefore, $E_y \equiv 0 \bmod 2$), then

$$\begin{aligned}
E &= (E_x - 1)/2 \\
x &= r^{2E} \cdot (rx_0 + x_1, x_2, \cdots, x_n, \cdots)_r \\
y &= r^{2E-1} \cdot (ry_0 + y_1, y_2, \cdots, y_n, \cdots)_r,
\end{aligned}$$

4) if $E_x \leq E_y$ and $E_y \equiv 0 \bmod 2$, then

$$\begin{aligned}
E &= E_y/2 \\
x &= r^{2E} \cdot \left(\underbrace{0, 0, \cdots, 0}_{E_y - E_x \text{ zeros}}, x_0, x_1, \cdots, x_n, \cdots \right)_r \\
y &= r^{2E-1} \cdot (ry_0 + y_1, y_2, \cdots, y_n, \cdots)_r,
\end{aligned}$$

5) if $E_x \leq E_y$ and $E_y \equiv 1 \bmod 2$, then

$$\begin{aligned}
E &= (E_y + 1)/2 \\
x &= r^{2E} \cdot \left(\underbrace{0, 0, \cdots, 0}_{E_y - E_x + 1 \text{ zeros}}, x_0, x_1, \cdots, x_n, \cdots \right)_r \\
y &= r^{2E-1} \cdot (y_0, y_1, \cdots, y_n, \cdots)_r.
\end{aligned}$$

All that now remains for us to do is convert the aligned sequences to radix r^2 (conversion from radix r to radix r^2 is rather straightforward), and alternate the signs of the weighted factors in accordance with (20).

Example 1 *Find a representation of* $z = 1234.5 + i \cdot 9.876$ *in radix* $10i$. *We have:*

$$x = 1.2345 \cdot 10^3 = 10^3 \cdot (1, 2, 3, 4, 5)_{10}$$
$$y = 9.876 \cdot 10^0 = 10^0 \cdot (9, 8, 7, 6)_{10}$$

After aligning the decimal points (here $E_x = 3$, $E_y = 0$, *therefore,* $E = (E_x - 1)/2 = 1$*):*

$$x = 10^2 \cdot (12, 3, 4, 5)_{10}$$
$$y = 10^1 \cdot (0, 9, 8, 7, 6)_{10}$$

In radix $r^2 = 100$*:*

$$x = 10^2 \cdot (12, 34, 50)_{100}$$
$$y = 10^1 \cdot (0, 98, 76)_{100}$$

Finally, using (20), we find

$$z = 100^1 \cdot (12, 0, -34, 98, 50, -76)_{ri}.$$

Example 2 *Find a representation of* $z = 9.876 + i \cdot 1234.5$ *in radix* $10i$. *We have:*

$$x = 9.876 \cdot 10^0 = 10^0 \cdot (9, 8, 7, 6)_{10}$$
$$y = 1.2345 \cdot 10^3 = 10^3 \cdot (1, 2, 3, 4, 5)_{10}$$

After aligning the decimal points (here $E_x = 0$, $E_y = 3$, *therefore,* $E = (E_y + 1)/2 = 2$*):*

$$x = 10^4 \cdot (0, 0, 0, 0, 9, 8, 7, 6)_{10}$$
$$y = 10^3 \cdot (1, 2, 3, 4, 5)_{10}$$

In radix $r^2 = 100$*:*

$$x = 10^4 \cdot (0, 0, 9, 87, 60)_{100}$$
$$y = 10^3 \cdot (1, 23, 45)_{100}$$

Using (20), we obtain a representation of z

$$z = 100^2 \cdot (0, -1, 0, 23, 9, -45, -87, 0, 60)_{ri}.$$

4.2 The selector functions

The two selector functions, Re and Im, are needed to extract the real and imaginary parts of complex values, respectively. The algorithms for the evaluation of the selector functions are essentially the inverse of those for the constructor function: given a radix-ri expansion

$$z = R^E \cdot (z_0 + z_1(ri)^{-1} + z_2(ri)^{-2} + \ldots + z_n(ri)^{-n} + \ldots),$$

the real and imaginary parts can be found as follows:

$$\operatorname{Re} z = r^{2E} \cdot \sum_{n=0}^{\infty} x_n \left(r^2\right)^{-n},$$
$$\operatorname{Im} z = r^{2E-1} \cdot \sum_{n=0}^{\infty} y_n \left(r^2\right)^{-n},$$

where

$$x_n = (-1)^n z_{2n}, \quad y_n = (-1)^{n+1} z_{2n+1}.$$

These expansions of $\operatorname{Re} z$ and $\operatorname{Im} z$ are in radix r^2; conversion from radix r^2 to radix r is can be performed with trivial effort.

5 Conclusions

We have introduced the redundant radix ri representation of exact complex numbers, and discussed the algorithms for the four basic arithmetic operations. The algorithms are to an unexpected extent similar to their exact real counterparts, which makes the theoretical complexity of operations on exact complex numbers comparable with that of the corresponding operations on exact reals. The author's implementation of the algorithms in the functional programming language Miranda[2] shows a clear speed advantage over conventional methods in executing multiplication and division, and equivalent performance in addition and subtraction.

The author does not know of any other implementations of exact complex arithmetic based upon an imaginary radix, and there are a number of possible directions for ulterior research. Of these, we are most interested in:

- Efficiency aspects of implementations. The complexity issues concerning operations on exact numbers for the most part remain unresolved.

[2]Miranda™ is a trademark of Research Software Limited

Some measurements have revealed that exact arithmetic performs 50 to 100 times slower than hardware floating point [25]. It is generally doubtful that exact real or complex arithmetic will ever perform on current architectures as speedily as floating point which was designed to match the underlying hardware, although one can think of various ways to optimize the existing implementations.

- Elementary function evaluation. The evaluation of elementary functions in positional radix systems is a very cumbersome process, because there are no obvious on-line digit-by-digit algorithms that are both simple and efficient. Almost the only commonly used elementary function that can be evaluated iteratively with little additional overhead beyond that of the basic arithmetic operations is the square root function. In radix-r systems, square root can be computed using a simple pseudo-division algorithm that produces n digits of the result in n cycles, at a rate of one digit per cycle. Since the only essential difference between square root and division is in the recurrence relation and the digit selection function, we may reasonably expect to find a similar square-rooting algorithm for complex numbers. Generally, where radix-r algorithms exist for the evaluation of elementary functions of a real variable, we may also expect to find their radix-ri equivalents.

  Many complex elementary functions, such as the exponential, hyperbolic and trigonometric functions seem to be unrelated inside the domains of their real restriction, but going to the complex plane reveals that these functions are in fact all intimately connected. For example, an algorithm that could compute the exponent of a radix-ri sequence of digits would also compute the trigonometric functions:

$$\cos z = \frac{e^{iz} + e^{-iz}}{2}, \quad \sin z = \frac{e^{iz} - e^{-iz}}{2i},$$

  as well as the hyperbolic functions

$$\cosh z = \cos iz, \quad \sinh z = -i \sin iz.$$

Acknowledgements

Support for this research was provided by EPSRC Grant Ref. GR/L03279. The author wishes to express his sincere and deep gratitude to Professor David Turner for his many suggestions and valuable comments concerning the exposition of these ideas.

A Miranda code for exact complex arithmetic

```
FILE:  complex.m
----------------
/*  Copyright (C) Alexander Kaganovsky 1995-1999.  All rights reserved. */

%export complex mc re im cadd csub cmul cdiv showc

%include "input.m"   || input validation functions
%include "real.m"    || exact real arithmetic package

abstype complex
with mc :: real -> real -> complex
     re, im :: complex -> real
     cadd, csub, cmul, cdiv :: complex -> complex -> complex
     showc :: num -> num -> complex -> [char]
     showcomplex :: complex -> [char]

complex == (num,[num])
real == (num,[num])

expo (e,m) = e
man (e,m) = m

||--------------------------------------------------------------------||
|| Conversion functions (base 10 to 100)                              ||
||--------------------------------------------------------------------||
cm10to100 :: [num] -> [num]
cm10to100 [] = []
cm10to100 (a:x)
  = (10*a+(hd x)) : cm10to100 (tl x),   if x ~= []
  = 10*a : cm10to100 x,                 otherwise

|| cman10to100 is the same as cm10to100, except it doesn't convert
|| the first element of the list
cman10to100 (a:x) = a:cm10to100 x
cman10to100 [] = []

||--------------------------------------------------------------------||
|| Conversion functions (base 100 to base 10)                         ||
||--------------------------------------------------------------------||
```

```
cm100to10 :: [num] -> [num]
cm100to10 [] = []
cm100to10 (a:x)
  = a0 : a1 : cm100to10 x
    where
    (a0,a1) = divrem a 10

cman100to10 (a:x) = a:cm100to10 x
cman100to10 [] = []

||-------------------------------------------------------------------||
|| mc converts two exact reals (base 10) to a single-component        ||
|| complex format (base 10i)                                          ||
||-------------------------------------------------------------------||
mc (ex,mx) (ey,my)
  = (ez, mz)
    where
    mz = alternate (cman10to100 mx') (cman10to100 my')
    (ez,mx',my')
        = (ex div 2, mx, inc (ex-ey-1) my),          if (ex > ey) &
                                                        (ex mod 2 = 0)
        = ((ex-1) div 2, dec mx, inc (ex-ey-2) my),  if (ex > ey+1) &
                                                        (ex mod 2 = 1)
        = ((ex-1) div 2, dec mx, dec my),            if (ex = ey+1) &
                                                        (ex mod 2 = 1)
        = (ey div 2, inc (ey-ex) mx, dec my),        if (ex <= ey) &
                                                        (ey mod 2 = 0)
        = ((ey+1) div 2, inc (ey-ex+1) mx, my),      otherwise

|| dec adjusts the mantissa of a radix-10 number to compensate for a dec of
|| (subtraction of one from) its exponent by multiplying the mantissa by 10
dec :: [num] -> [num]
dec [] = []
dec [a] = [10*a]
dec (a:x) = (10*a+(hd x)):(tl x)

|| inc adjusts the mantissa of a radix-10 number to compensate for an
|| increase of its exponent by n.  It does so by dividing the mantissa
|| by 10^(-n), i.e. adding n leading zeros
inc :: num -> [num] -> [num]
inc 0 x = x
```

```
inc n x = zeros n ++ x
zeros n = rep n 0

||-----------------------------------------------------------------||
|| re returns the real part (represented in base 10) of a complex ||
|| number (base 10i), im - the imaginary part of same             ||
||-----------------------------------------------------------------||
re (e,x) = can (2*e, cman100to10 (even x))
im (e,x) = can (2*e-1, cman100to10 (odd x))

||-----------------------------------------------------------------||
|| alternate (x0,x1,...) (y0,y1,...) = (x0,-y0,-x1,y1,x2,-y2,...)  ||
||-----------------------------------------------------------------||
alternate :: [num] -> [num] -> [num]
alternate [] [] = []
alternate [] [y0] = [0,-y0]
alternate [] [y0,y1] = [0,-y0,0,y1]
alternate [x0] [] = [x0]
alternate [x0] [y0] = [x0,-y0]
alternate [x0] [y0,y1] = [x0,-y0,0,y1]
alternate [x0] (y0:y1:ys) = [x0,-y0,0,y1] ++ alternate [0] ys
alternate [x0,x1] [] = [x0,0,-x1]
alternate [x0,x1] [y0] = [x0,-y0,-x1]
alternate (x0:x1:xs) [y0] = [x0,-y0,-x1,0] ++ alternate xs [0]
alternate (x0:x1:xs) (y0:y1:ys)
  = [x0,-y0,-x1,y1] ++ alternate xs ys

||----------------------------------------------------------------||
|| even (z0,z1,z2,...) = (z0,-z2,z4,...)                          ||
|| odd  (z0,z1,z2,...) = (-z1,z3,-z5,...)                         ||
||----------------------------------------------------------------||
even z = skipevenodd 0   1  z
odd  z = skipevenodd 1 (-1) z

skipevenodd :: num -> num -> [num] -> [num]
skipevenodd counter sign (z0:z)
 = (sign*z0):(skipevenodd 1 (-sign) z),        if counter mod 2 = 0
 = skipevenodd 0 sign z,                       otherwise
skipevenodd counter sign [] = []
```

```
||---------------------------------------------------------------------||
||                    Complex normalization functions                  ||
||---------------------------------------------------------------------||
creduce z = alternate (reduce (even z)) (reduce (odd z))

ccan (e,m)
  = (e,m),              if m=[]
  = (e,creduce m),       otherwise

||---------------------------------------------------------------------||
||                           Complex addition                          ||
||---------------------------------------------------------------------||
cadd (ez,mz) (ew,mw)
  = ccan (ez,m),                if ez >= ew
  = cadd (ew,mw) (ez,mz),       otherwise
    where
    m = add mz mw,                                     if ez = ew
      = add mz (zeros (2*(ez-ew)) ++ mw),              if (ez-ew) mod 2 = 0
      = add mz (zeros (2*(ez-ew)) ++ (map neg mw)),   otherwise

add (a:x) (b:y) = a+b : add x y
add x [] = x
add [] y = y

||---------------------------------------------------------------------||
||                         Complex subtraction                         ||
||---------------------------------------------------------------------||
csub (ez,mz) (ew,mw) = cadd (ez,mz) (ew, negate mw)


||---------------------------------------------------------------------||
||                        Complex multiplication                       ||
||---------------------------------------------------------------------||
cmul z w
  = ccan (e,m)
    where
    e = expo z + expo w
    m = mantissa_mult (man z) (man w)
    mantissa_mult zs ws
      = [],                              if zs=[] \/ ws=[]
      = mult_reduce cross_products_list,  otherwise
```

```
        where
        cross_products_list = [map (* t) ws | t<-zs]
        mult_reduce z = [],                                             if z=[[]]
            = take 228 block ++
               mult_reduce (drop 228 block:drop 229 z), otherwise
              where
              block = (creduce.creduce.diag_add.take 229) z
              diag_add (a:b:x)
                = [],                                   if a=[]
                = a!0 : diag_add (add (tl a) b : x),  otherwise
              diag_add [a] = a
              diag_add [] = []

||----------------------------------------------------------------------||
||                         Complex division                             ||
||----------------------------------------------------------------------||
cdiv z w
  = (ccan.kill_zeros 4) (exponent, repdiv (man z))
    where
    exponent = expo z - expo divisor
    divisor = norm_divisor (kill_all_zeros w)
    w0:ws = man divisor
    repdiv z
      = [],                 if z=[]
      = nextdigit:quotient, otherwise
        where
        nextdigit = guess (z!0) ((z!1) div 10) w0 ((ws!0) div 10)
        quotient = repdiv (creduce (times_ri m))
        m = add z (map (*(-nextdigit)) (w0:ws))
        guess z0 z1 w0 w1 = fst (divrem (z0*w0+z1*w1) (w0*w0+w1*w1))

|| kill zeros
kill_zeros n (e,m)
  = kill_zeros (n-2) (e-1,map neg (tl (tl m))), if (tl m)~=[] & m!0=0 &
                                                   m!1=0
  = (e,m),                                        otherwise

kill_all_zeros = kill_zeros (-1)

add_leading_zero (e,m) = (e,0:m)
```

```
|| normalize the divisor
norm_divisor (e,m)
  = error "attempt to divide by zero",                  if m=[]
  = norm_divisor (e-1,map neg ((times_ri.times_ri) m)),
                    if (abs (m!0) < 10000) & (abs (m!1) < 10000)
  = (e,m),                                              otherwise

|| times_ri multiplies a list by 10i
times_ri [] = []
times_ri [0] = []
times_ri [z0] = [0,-100*z0]
times_ri [z0,z1] = [z1,-100*z0]
times_ri [z0,z1,z2] = [z1,z2-100*z0]
times_ri (z0:z1:z) = z1:((hd z)-100*z0):(tl z)


||--------------------------------------------------------------------||
|| The output function:  "showc nr ni z" prints (nr) radix-10 digits of ||
|| the real part and (ni) radix-10 digits of the imaginary part of z    ||
|| Requires the "st" function from the exact real arithmetic package    ||
||--------------------------------------------------------------------||
showc nr ni z
    = (st nr (re z)) ++ " + i * " ++ (st ni (im z))

|| show only 10 digits by default
showcomplex z = showc 10 10 z
```

References

[1] Lang, S., *Algebra*, (3rd Ed.), Addison-Wesley Pub. Co., 1992

[2] *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), by W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, Cambridge University Press, 1995

[3] Muller, J.-M. "Arithmétique des ordinateurs", *Etudes et recherches en informatique*, Masson, 1989

[4] Kahan, W., "Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit", Chapter 7 in *The State of the Art in Numerical Analysis*, ed. by M. Powell and A. Iserles, Oxford (1987), pp. 165-211

[5] Boehm, H.-J., Cartwright R., et al., "Exact Real Arithmetic: A Case Study in Higher Order Programming", *Proceedings 1986 ACM Conference on LISP and Functional Programming*, ACM Press (August 1986), pp. 162-163

[6] Boehm, H. and Cartwright, R., "Exact Real Arithmetic: Formulating Real Numbers as Functions", in *Research Topics in Functional Programming*, (ed) D. A. Turner, Addison-Wesley, 1990

[7] Wiedmer, E., "Computing with Infinite Objects", *Theoretical Computer Science*, Vol. 10 (1980), pp. 133-155

[8] Vuillemin, J. E., "Exact Real Computer Arithmetic with Continued Fractions", *IEEE Transactions on Computers*, Vol. 39, No. 8, August 1990, pp. 1087-1105

[9] Edalat, A. and Potts, P. J., "A New Representation for Exact Real Numbers", Electronic Notes in Theoretical Computer Science, 6 (1997)

[10] Kaganovsky, A., "Computing with Exact Real Numbers in a Radix-r System", Electronic Notes in Theoretical Computer Science, 13 (1998)

[11] Knuth, D. E., "An Imaginary Number System", *Commun. ACM*, Vol. 3, No. 4, Apr. 1960, pp. 245-247

[12] Penney, W., "A 'Binary' System for Complex Numbers", *Jl ACM*, Vol. 12, No. 2, Apr. 1965, pp. 247-248

[13] O'Reilly, T. J., "A Positional Notation for Complex Numbers", *IEEE Computer Society Repository, R74-169*, 12 pp. (1974), IEEE Computer Society Publications Office, Long Beach, CA 90803

[14] Holmes, W. N., "Representation for Complex Numbers", *IBM Jl Res. Develop.*, Vol. 22, No. 4, July 1978, pp. 429-430

[15] Wadel, L. B., "Negative Base Number Systems", *IRE Trans. Electr. Comp.*, Vol. EC-6, June 1957, p. 123

[16] Songster, G. F., "Negative-Base Number-Representation Systems", *IEEE Trans. Electr. Comp.*, June 1963, pp. 274-277

[17] de Regt, M. P., "Negative Radix Arithmetic", *Comp. Design*, Vol. 16, May 1967, pp. 52-63

[18] Sankar, P. V., Chakrabarti, S., and Krishnamurthy, E.V., "Arithmetic Algorithms in a Negative Base", *IEEE Trans. Comput.*, Vol. C-22, No. 2, Feb. 1973, pp. 120-125

[19] Pawlak, Z., "An Electronic Digital Computer Based on the '—2' System", *Bull. de l'Acad. Polonaise des Sciences, Série des Sci. Tech.*, Vol. 7, No. 12, 1959, pp. 713-721

[20] Avizienis, A., "Signed-digit number representations for fast parallel arithmetic", *IRE Trans. El. Comp.*, Vol. EC-10, No. 3, Sept. 1961, pp. 389-400

[21] Yuen, C. K., "On the Floating Point Representation of Complex Numbers", *IEEE Trans. Comput.*, Vol. C-24, No. 6, Aug. 1975, pp. 846-848

[22] Stromberg, K. R., *Introduction to Classical Real Analysis*, Wadsworth, Inc., 1981

[23] Robertson, J. E., "A New Class of Digital Division Methods", *IRE Trans. El. Comp.*, Vol. EC-7, Sept. 1958, pp. 218-222

[24] Smith, D. M., "A Multiple-Precision Division Algorithm", *Mathematics of Computation*, Vol. 65, No. 213, Jan. 1996, pp. 157-163

[25] Schwarz, J., "Implementing Infinite Precision Arithmetic", *Proc. 9th Symposium on Computer Arithmetic*, September 6-8, 1989, pp. 10-17